12 Factor Applications

with Docker and Go

12 Factor Applications with Docker and Go

A book filled with examples on how to use Docker and Go to create the ultimate 12 Factor applications

Tit Petric

This book is for sale at http://leanpub.com/12fa-docker-golang

This version was published on 2018-02-24

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2018 Tit Petric

Tweet This Book!

Please help Tit Petric by spreading the word about this book on Twitter!

The suggested hashtag for this book is #12fadocker.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#12fadocker

Contents

Introduction **1**
About me 1
Who is this book for? 1
How should I study it? 2

Requirements **3**
Linux and Docker 3
Additional requirements 6

I. Codebase - One codebase tracked in revision control, many deploys **9**
Gogs 10
Multiple deploys 13
Structuring your code 16
Improving deployments 18

II. Dependencies - Explicitly declare and isolate dependencies **20**
Vendoring 20
Caveats about vendoring 21
System dependencies 23

III. Config - Store config in the environment **26**
Flags 27
godotenv 31
Viper 32

IV. Backing services - Treat backing services as attached resources **35**
Data resources 36
Example backing services 38

V. Build, release, run - Strictly separate build and run stages **50**
Building Go applications 51
Building Docker images 53
Continuous integration with Codeship 54

VI. Processes - Execute the app as one or more stateless processes . . . 65
Data caching . . . 66
Share-nothing . . . 66
Backing service: Running a highly available Redis instance . . . 67
Realistic example of a stateless application . . . 69

VII. Port binding - Export services via port binding . . . 72
The none network . . . 73
The bridge network . . . 74
Custom bridge network . . . 75
Multi-host network . . . 76
Host networking . . . 78
Exposing your service ports . . . 79
Debugging docker network connectivity . . . 81
Using your own DNS . . . 84
Sending e-mails from docker . . . 86

VIII. Concurrency - Scale out via the process model . . . 88
Basic process management . . . 88
Advanced process management . . . 90
Docker swarm . . . 95
Scaling our service with Docker Swarm . . . 104
Rolling updates for our service . . . 110
Scaling to a 1000 containers! . . . 112

IX. Disposability - Maximize robustness with fast startup and graceful shutdown . . . 115
Disposing of containers . . . 115
Docker Chaos Monkey . . . 118
Health checks . . . 120

X. Dev/prod parity - Keep development, staging, and production as similar as possible . 122
It worked on my machine . . . 123
Upgrades and testing . . . 124
Many moving parts . . . 126

XI. Logs - Treat logs as event streams . . . 127
A note on backing services . . . 128
Logging Docker to Papertrail . . . 129
Logging as a backing service . . . 133

XII. Admin processes - Run admin/management tasks as one-off processes . . . 136
Running backups . . . 137
Console access to your containers . . . 139

Orchestrating your management tasks 142

Introduction

About me

I'm one of those people with about two decades of programming experience under my belt. I've started optimizing code in the 90's, discovered PHP in the 2000's, and build several large-scale projects on to the day, all while discovering other programming language families like Java, Node.js and ultimately, Go.

I have built numerous APIs for my own content management products. Several products I've been involved with as a lead developer have been sold and are used in multiple countries. I've written a professional dedicated API framework which doubles as an software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. I'm also the speaker at several local PHP user group events and conferences.

I am also the author of API Foundations in Go[1].

I write a blog which is available on scene-si.org[2], you should check it out, I tend to publish new articles about twice monthly.

Who is this book for?

This book is for everyone who writes applications for a living. I cover a wide area of subjects dedicated to development of software in general, trying to establish some good development practices, while at the same time referencing the 12 Factor App manifesto.

In the book, I will cover these subjects:

- I. Codebase - One codebase tracked in revision control, many deploys
- II. Dependencies - Explicitly declare and isolate dependencies
- III. Config - Store config in the environment
- IV. Backing services - Treat backing services as attached resources
- V. Build, release, run - Strictly separate build and run stages
- VI. Processes - Execute the app as one or more stateless processes
- VII. Port binding - Export services via port binding
- VIII. Concurrency - Scale out via the process model

[1] https://leanpub.com/api-foundations

[2] https://scene-si.org

- IX. Disposability - Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity - Keep development, staging, and production as similar as possible
- XI. Logs - Treat logs as event streams
- XII. Admin processes - Run admin/management tasks as one-off processes

Covering these concepts should give you a good overview of the 12 Factor App designs and what problems they are trying to solve. The book tries to give you a set of best practices to follow and guide you through individual chapters shedding light and hands on examples of individual approaches.

How should I study it?

Through the book, I will present several examples on how to do common things when developing APIs. The examples are published on GitHub[3].

You should follow the examples in the book, or you can look at each chapter individually, just to cover the knowledge of that chapter. The examples are stand-alone, but generally build on work from previous chapters. Be sure to follow the Requirements section as you're working with the book.

[3] https://github.com/titpetric/books

Requirements

This is a book which demonstrates 12 factor application development on the use case of developing and deploying microservices. As such, there are a few prerequisites that need to be taken care of, when you'll be going through the examples in the book.

Linux and Docker

The examples of the book rely on a recent docker-engine installation on a Linux host.

Own hardware

The recommended configuration if you have your own hardware is:

- 2 CPU core,
- 2GB ram,
- 128GB disk (SSD)

The minimal configuration known to mostly work is about half that, but you might find yourself in a tight place as soon as your usage goes up. If you're just tying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

If you're running windows 10, you can enable the Hyper-V service, which allows you to install and run Linux in a virtual machine on your laptop/PC, I highly recommend this option.

Please refer to the official docker installation instructions[4] on how to install a recent docker version.

Cloud quick-start

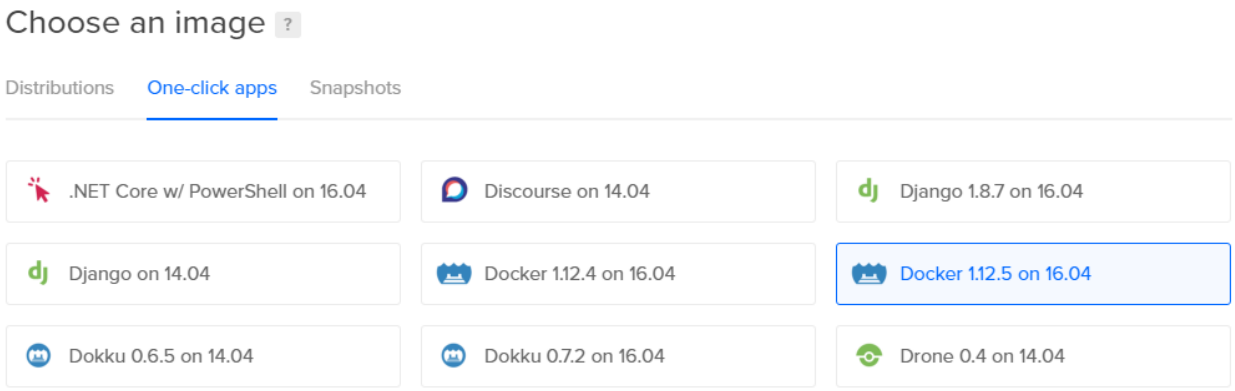
If having your own hardware is a bit of a buzzkill, welcome to the world of the cloud. You can literally set up your own virtual server on Digital Ocean within minutes. You can use this DigitalOcean referral link[5] to get a $10 credit, while also helping me take some zeros of my hosting bills.

After signing up, creating a Linux instance with a running Docker engine is simple, and only takes a few clicks. There's this nice green button on the top header of the page, where it says "Create

[4] https://docs.docker.com/engine/installation/linux/

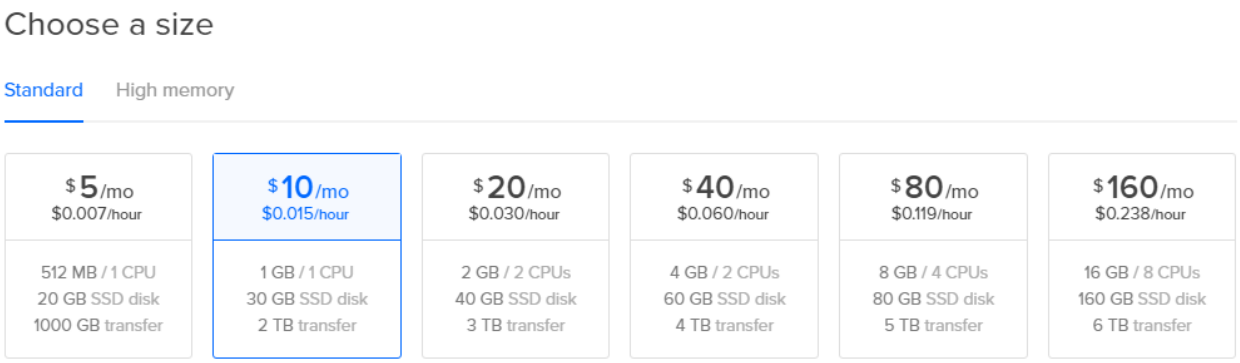
[5] https://m.do.co/c/021b61109d56

Droplet". Click it, and on the page it opens, navigate to "One-click apps" where you choose a "Docker" from the list.



Choose Docker from "One-click apps"

Running docker can be disk-usage intensive. Some docker images may "weigh" up to or more than 1 GB. I would definitely advise choosing an instance with *at least* 30GB of disk space, which is a bargain for $10 a month, but you will have to keep an eye out for disk usage. It's been known to fill up.



Choose a reasonable disk size

Aside for some additional options on the page, like chosing a region where your droplet will be running in, there's only a big green "Create" button on the bottom of the page, which will set up everything you need.

Creating Digital Ocean droplets via API

Digital Ocean provides the CLI utility doctl[6]. It's a Go program that interfaces with their API to allow you to list running droplets, create new ones, shut them down and much more. If you don't need an instance 24/7, you can use the API to spin it up and shut it down based on your needs. It's

[6] https://github.com/digitalocean/doctl

good for having better quality development machines, which you delete after you're done for the day, saving about half of the costs.

Download and install `doctl` from the releases page and issue `docker auth init` to authenticate against the Digital Ocean API. You can generate the token from the top of your dashboard.

```
# doctl auth init
DigitalOcean access token: 24df63816084cb8270fbe03a7c98341a
Validating token: OK
```

After this, you're free to list images and create new droplets via the API.

```
# doctl compute image list | grep docker
23219707    Docker 17.03.0-ce on 14.04          docker
24232340    Docker 17.04.0-ce on 16.04          docker-16-04
```

We can use the `docker-16-04` image to quickly spin up a droplet. We will create `create.sh` and `destroy.sh`, which will check for existance of the droplet before it's created or destroyed.

Create a new droplet conditionally

```
#!/bin/bash
CHECK=$(doctl compute droplet list asx --format ID --no-header | wc -l)
if [ "$CHECK" == "0" ]; then
        echo "Creating ASX droplet"
        doctl compute droplet create asx -v \
                --image docker-16-04 \
                --size 2gb \
                --region ams3 \
                --ssh-keys $(./ssh-key.sh)
else
        echo "Droplet ASX already running"
fi
```

You should change the parameter to `--ssh-keys` with either your SSH key or a SSH key fingerprint, once you've added the SSH key to the Digital Ocean settings page under "Security". You can adjust the parameters based on the size of the droplet you would like and the region where you want it to run.

Destroying the droplet is just as simple as creating one:

Destroy a droplet if it exists

```
#!/bin/bash
CHECK=$(doctl compute droplet list asx --format ID --no-header | wc -l)
if [ "$CHECK" == "0" ]; then
        echo "Droplet ASX not started"
else
        echo "Deleting droplet asx"
        doctl compute droplet delete asx -f -v
fi
```

Of course, when your droplet spins up, you'll want to list it's public IP, or connect to it via SSH.

Listing the public ip for a droplet

```
doctl compute droplet list asx --format PublicIPv4 --no-header
```

Connecting to the IP is just as simple as wrapping everything in a `ssh $( + )`.

Connect to the droplet

```
ssh $(doctl compute droplet list asx --format PublicIPv4 --no-header)
```

As soon as you SSH into the instance, you can use any `docker run` command you like. At this point it's up to you to set up any git checkouts or somehow populate the running droplet.

All the scripts to create Digital Ocean instances programmatically are available in the book samples GitHub repository[7].

Additional requirements

Networking

When dealing with microservices in docker, it's a very common practice to enable communication between one service and another. For example, many microservices may and do communicate to each other. For this purpose we create a network called "party", on which the microservices from this book will run on.

[7] https://github.com/titpetric/books/tree/master/12fa-docker-golang/digitalocean

Create a custom bridge network

```
#!/bin/bash
#
# this creates a custom docker "bridge" network named "party"
#
docker network create -d bridge --subnet 172.25.0.0/24 party
```

When running microservices, it is enough to join the docker container to this network with the option `--net=party`. You may define multiple networks to isolate docker containers away from each other.

MySQL database

Some services will require a MySQL database to store data. We can quickly start up an instance in docker as well. We will name our instance "mysql" and will try to re-use it when ever an opportunity arises.

Quickstart: Run a MySQL instance

```
#!/bin/bash
NAME="mysql"
APP_DIR=$(dirname $(readlink -f $0))
mkdir -p ${APP_DIR}/data/$NAME
docker stop $NAME
docker rm $NAME

DOCKERFILE="titpetric/percona-xtrabackup"

docker stop $NAME
docker rm $NAME
docker run --restart=always \
    -h $NAME \
    --name $NAME \
    --net=party \
    -v ${APP_DIR}/data/$NAME/data:/var/lib/mysql \
    -e MYSQL_ALLOW_EMPTY_PASSWORD="yes" \
    -d $DOCKERFILE
```

When using the instance from other docker containers on the same network, you may use the following credentials to connect to it:

- hostname: `mysql` (the same as the container name),
- username: `root`,
- password: empty string

> Note: obviously having a MySQL instance with an empty root password in production is bad form. If you'd like to run MySQL with Docker in production, take a look at the default options in the percona image[a] on Docker Hub. There's a slightly better example of using MySQL in chapter 4 (backing services). Please consult that chapter for more examples.
>
> [a] https://hub.docker.com/_/percona/

I. Codebase - One codebase tracked in revision control, many deploys

Source code VCS (version control systems) are designed to store revisions of your source code. For each feature you add, they log a "commit" or version, which is a point in time snapshot of your changes. Commits can be recalled at a later time in case you need to roll-back your code for any reason, including getting a deleted section of source code restored at a later time.

Commits, as they are only partial snapshot on the changes on your code base, are an ideal way to enable collaborative work on software projects - if user 1 edits file 1 and user 2 edits file 2, there can be no conflict when merging commits between these two users. And when it comes to editing the same file - users work on local copies, and when they push their commits, a conflict might occur. Resolving the conflict may be as simple as editing a file, picking the lines which should stay and committing the changes.

I'd suggest you use `git` for revision control. You can use services like GitHub[8] or Bitbucket[9], or host your own with GitLab[10] or Gogs[11]. Make sure that you perform a full backup on occasion and have procedures in place which allow you to restore it from a backup. This should go without saying.

For company use, I'd recommend starting with a Bitbucket, which is free up to 5 users, and allows unlimited private repositories. If you want more than the 5 free users, the pricing is $10/month for 10 users, $20/month for 20, and so on.

If you're developing for/with community, I'd definitely recommend you to use GitHub instead. While pricing for private repos is higher, GitHub also provides a coherent issue tracker, lots of tooling to support your development, and an invaluable community. A team account with 5 users starts with $25/month.

Your application project or microservice should use one repository just for the sake of simplicity. You can use multiple repositories in a project, but it will increase complexity, usually not with good results. You can always structure your Go application in subpackages and keep everything neatly organized. If you are making several applications that use the same parts of code, this code should be separated into packages which may be included via `go get` or preferably a dependency manager (See chapter 2).

Let's to through setting up a self-hosted git service, where we can start work right away.

---

[8] https://github.com
[9] https://bitbucket.org
[10] https://gitlab.com
[11] https://gogs.io

Gogs

Gogs is a great option if you want a zero-cost self-hosted git solution. I'm going to walk you through the process of setting up Gogs. And since this is a book that relies heavily on docker, we'll use it to set up our Gogs instance.

Why Gogs in the first place?

As mentioned, there are a lot of services that are the standard today, most notably GitHub and Bitbucket. Bitbucket is a great option if you want a free plan that allows private repositories and managing permissions in a small team of up to 5 people, and they give you three extra people if you invite them to sign up. But, setting up your own git repositories gives you the benefit of avoiding things like rate limiting or issues with these services. If you're fanatic about keeping your code off the cloud, hosting it yourself is also a good option (but don't skip the section on backing it up). If your needs tend to grow (as they do), it's always an easy task to migrate from Gogs to GitLab, GitHub or Bitbucket at a later time, and I'm going to show you how to do that as well.

Installing and running Gogs

Gogs is written in Go. This means, they have a nice little set of binaries available for most common distributions and architectures, including Windows, if you like. But as we like the comfort of a process manager around Gogs, we will run it in docker. I've created a script to do the following for convenience:

Quick start: Run a Gogs instance

```
#!/bin/bash
NAME="gogs"
APP_DIR=$(dirname $(readlink -f $0))
mkdir -p ${APP_DIR}/data/$NAME
docker stop $NAME
docker rm $NAME
docker run -d --net=party --restart=always --name=$NAME -h $NAME -p 10022:22 -p \
3000:3000 -v ${APP_DIR}/data/$NAME:/data gogs/gogs
```

The process manager part here is the `--restart` option, which instructs docker to restart the container should it exit for any reason. A possible reason being, that the server was rebooted for maintenance. After executing this script, Gogs is running in a docker container with the same name, exposing ports 10022 for ssh access, and port 3000 for http access.

When you open `http://yourserver:3000/` an install process will guide you through the installation. In the options, you may configure it by connecting to the MySQL instance we created in the

Introduction chapter, or you may use a `SQLite3` back-end, to store it's data locally without a database server.

In case you ran the `mysql` service from the Introduction chapter, you will need to create a `gogs` database, which will store the table schema and data for the Gogs application:

```
1 docker exec mysql mysqladmin create gogs
```

> Note: When it comes to production use, individual usernames should be created for individual microservices, only giving access to the database of the microservice. That way, a `gogs` user could only access the database with the same name. If a microservice needs data from another database, it should reasonably get this data from another microservice (Consult Chapter IV. Backing Services).

Logging in

Just in case if you didn't enter an administrative user in the installation section: The first user who will be registered in Gogs, will become the administrative user. There's no default user created.

Creating a repository

We can create a new repository by clicking the "+" next to "Repositories" (on the right side). As a test we can create a 'the12factors' repository. Gogs will give you nice instructions how to create a new repository or push an existing repository to it. For our purposes, we will create our own repository:

```
1 mkdir the12factors
2 cd the12factors
3 touch README.md
4 git init
5 git add README.md
6 git commit -m "first commit"
7 git remote add origin http://peer.lan:3000/titpetric/the12factors.git
8 git push -u origin master
```

The repository will be created and new changes available

> Note: every git push will ask for username if you use the http method. It's possible to save these credentials on the system, so it will not ask you for them every time. A better method is to use the `ssh` endpoint, where authentication is done with an ssh key.

Backing up Gogs

If you set up Gogs with an SQLite database as suggested, the backup of Gogs is very simple:

Quick start: Backup a full Gogs instance

```
#!/bin/bash
NAME="gogs"
docker stop $NAME
cd data && tar -zcvf gogs-backup-$(date +"%Y%m%d").tgz gogs/ && cd ..
docker start $NAME
```

The backup script stops the Gogs instance, so all the data is written to disk, and then creates a backup archive with the current data in the `data/` folder. This data can be archived, or you can use it for migration to another service.

> Note: be sure to back up your MySQL instance in case you used it to set up Gogs. The database contains all the schema and data you will need to restore it on another server. You may use the same principle for backing up mysql data as we're doing with Gogs. These backup files will be the raw backup files that MySQL stores on disk, so you will need to pay attention that you're restoring the data into a compatible MySQL version, the same version if possible.

Multiple deploys

When you're are using git for version control, there are a few accepted methodologies of how to set up your code so that you can have manageable deploys to several environments: development, staging, testing, production.

The simplest workflow to implement is just to have a master branch, and whatever lives there is ready for production. New development features should be done on other branches. Ideally, the master branch would be read-only, and feature branches should be merged with pull requests. Pull requests enable code reviews, commenting and discussion before accepting some change into production.

Ideally what would happen with multiple developers:

Fork, clone and branch the repository

Depending on your project, you might want to enforce forking a sensitive repository, which you don't want trashed by developers missteps. A fork is a literal copy of the complete contents of a git repository. This fork is usually created via the web interface on GitHub, GitLab, Bitbucket, Gogs and other git repository services. Each developer should have read-only access to the main source repositories.

If you don't enforce forks, you should enforce a read-only master branch. GitHub as well as Bitbucket allow you to do this, most likely GitLab and Gogs as well. Consider this something like a contract between your senior and junior developers. You should definitely enforce some limitations on the master branch that would prevent history rewrites and deletion, the logical conclusion is to have in read-only so that bad things could only happen on branches, which can effectively be deleted later.

Creating a branch on your cloned repository is as simple as running one command:

```
# git checkout -b new-feature
Switched to a new branch 'new-feature'
```

and if you want to use an existing branch, just skip the `-b` option. Inspect which branch you're working on by running `git branch`:

```
# git branch
  master
* new-feature
```

All the changes which you will now commit with git will be created in this branch. When using `git push` only the current branch will be pushed.

Squashing your changes

If your feature required several commits to finish, you might want to squash it into one commit, before you push the changes upstream. What this does is perform a local merge which will create a single commit instead of several which you choose. You can perform the squash by running:

```
# git rebase -i HEAD~2
```

Where the number 2 is the amount of commits you want to squash. You will be given a prompt with something similar to the following:

```
pick 255ac52 add title to readme
pick 3e9f63e add contents
```

And to actually squash the commits, you would change the trailing “pick” lines into “squash”:

```
pick 255ac52 add title to readme
squash 3e9f63e add contents
```

and then modify your commit message accordingly. The command will finish with something like this:

```
# git rebase -i HEAD~2
[detached HEAD cd3041c] add title to readme add contents
 Date: Mon Apr 17 08:16:00 2017 +0200
 1 file changed, 3 insertions(+)
Successfully rebased and updated refs/heads/new-feature.
```

We can inspect that the individual histories no longer exist:

```
# git log --oneline
cd3041c add title to readme add contents
525f2c9 import readme
```

Only the squashed commit is available on the branch now and you can push the changes to your branch.

Pushing changes

Pushing changes is as simple as issuing a `git push`. If you already pushed your commits, you will have to use a forced push to your branch:

```
# git push origin my-feature --force
```

Keep in mind, just like with the `master` branch, this is quite dangerous. If you're relying anywhere on `git pull`, this will break that functionality. When you rewrite history, it will break all the checkouts that already know about the commits which you're rewriting. For those, you will have to clone the repositories anew.

After you push your changes, a PR can be submitted to merge your changes to the master branch. These PRs are a good opportunity where you can introduce code reviews and automated testing. More about that in Chapter V.

Git fork/branch simplified

1. Create a fork
2. Clone the repository
3. Create a branch
4. Commit, squash and push changes
5. Submit a pull request

Other

I'm a proponent of feature branches, but there are other branching strategies available for you, depending on your product requirements. You may consider a *release branching* model, where work is done on branches which follow some naming convention, like semantic versioning.

> Note: More information about semantic versioning can be found on semver.org[a].
>
> [a] http://semver.org/

When your application and team get bigger, you should consider extending this practice with Git-Flow[12] or pick and choose a branching strategy that works for you.

[12] http://nvie.com/posts/a-successful-git-branching-model/

Structuring your code

As we will be using the Docker image `golang` for our examples, I will now explain how your project may be laid out, by explaining some things that go assumes about how you lay out your code. When you will use native vendoring, it means that you can continue to use `go run` and `go build`.

The GOPATH environment is set to `/go`. Your package layout should be something like this:

```
$GOPATH(/go)/src/app
  |__ main.go (Where your main code lives)
  |__ subpackage/
  |     |__ subpackage.go (You can create subpackages)
  |     \__ ...
  |__ ... (Other files or subpackages)
  \__ vendor/
        |__ github.com/namsral/flag
        |__ github.com/gorilla/mux
        \__ ...
```

Using subpackages is as simple as using `import "app/subpackage"`, where `app` is the folder name where your application is checked out. It's also possible to use a fully qualified domain name for the packages which are imported. In this case, the packages are installable in other apps, via commands like `go get` or vendoring tools like `godep` or `gvt`. More about this later.

Importing subpackages to your application

```
package main

import "app/apiservice"

func main() {
  apiservice.HelloWorld()
}
```

With our docker example, we use `app` for our project name. Obviously, this is up to you - but as you're creating a docker container for each run, we put whatever folder you have into `/go/src/app`.

Running your app with go run and docker

```
#!/bin/bash
docker run --rm -it -v `pwd`:/go/src/app -w /go/src/app golang go run main.go
```

When structuring the code like this, there are a few things I'd like to point out.

Relative imports

Don't use relative imports. In your main.go instead of `import "app/subpackage"` you could use `import "./subpackage"`, but it would break functionality. Relative imports at the time of writing don't work well with `vendoring`, which is a required step when building 12 factor apps (define and isolate dependencies).

Using subpackages externally

If we have multiple subpackages, let's say `subpackage1` and `subpackage2`, we shouldn't import these from subpackages with the same pattern. It will work for your application, but it will not work for an application which would use your subpackages with an import from a VCS.

Example of a problematic subpackage

```
1  package subpackage1
2
3  import "fmt"
4  import "app/subpackage2"
5
6  func Hello() {
7    return subpackage2.Hello();
8  }
```

We would use this subpackage from another application with something similar to this:

```
1  package main
2
3  import "example.github.com/username/your-app/subpackage1"
```

When you're doing this, you will have a problem that your new application knows nothing about "subpackage2", which is referenced in "subpackage1". There are only a few ways to fix this:

1. Use fully-qualified notation when importing other packages from subpackages,
2. Your packages should be self-contained (no importing subpackages from parent project)

There are various ways to achieve independence from other subpackages, while at the same time having a decoupled relationship with them. The most common way might be to use some sort of dependency injection. With this approach, your submodules would not be bound to other submodules (coupling), but would define an interface which can be satisfied with another submodule (loose coupling).

Improving deployments

When you're starting out with deployments, it's a very good idea to enforce some automation from the very beginning. Using a CI system is a must-have.

People use various continuous integration tools like Jenkins[13] or Travis[14], or cool hosted services like Buildkite[15] or Codeship[16]. It's not very hard to make tests when you're just starting to write your application, it's much harder when your application is already operational. Add tests in your CI workflow, so you don't publish any project which has tests that are failing.

The simplest way to test your Go application is running `go test`, but this is only one part of the testing which you need to pay attention to. If you're building a Docker image, it's a good idea that you set up a testing procedure for it as well.

Make sure that you can perform your build in one step. Usually it's enough to include a "build.sh" at the root of your project and then use any CI suite to run it. Depending on the size, this may invoke tools like `ant` or `make`, but when we're talking about microservices, we're talking about small applications that have a single responsibility, and not a big application that handles everything from data storage, session storage, templating, separate functionality for individual business areas. Ideally, each of these would be represented as a single microservice, so it can be replaced or optimized to take advantage of new underlying technology used.

Your testing procedures *must* be a condition of deploying an application. If the tests don't pass, the person who triggers the deploy has to be notified. It doesn't really matter who fixes the application or tests or when the fix is made, but until it's done - no deployments should be possible. This doesn't have to be a critical scenario in the sense of urgency to fix it right away, but can be done within weekly sprints.

That being said, the goal is to have individual microservices be deployable at any given moment, and a gold rule of thumb for sprints is that you really shouldn't deploy them on fridays, or on days where critical members of your team leave for vacation. There's a thing to be said on catching deadlines, but also planning on time to fix any issues which might occur after deployments and were not caught by tests.

Testing *may* be done on the development deployment, but care should be taken if the tests should be performed on every commit. There are several factors, which may be disruptive:

1. A developer may forget or ignore installing client-side git hooks,
2. The tests may be long-running and would prohibit a high-volume git commit workflow

It's recommended to move the testing to the server side and test after some code is pushed. Again, if a failing test is detected - it shouldn't be a cause for stopping work, it should just be a cause

[13] https://jenkins.io
[14] https://travis-ci.org/
[15] https://buildkite.com
[16] https://codeship.com

of not performing a deployment. Developers should have the capability of running tests on their development environment, give them this option by providing an individual “test.sh” or some documented way to do this.

With a Go project, this might be as simple as running `go test` but your application may test other things as well - like functionality with modifying database values. These tests can be more elaborate and use Docker for setting up a database container, loading testing data and then running parts of your application as if they were in a production system.

> Keep a practice of not assigning blame - when something breaks, just assign who is responsible for a fix. You should keep accountability while removing blame. Keep your team involved, so you may foster a team culture in resolving these issues. With individual issues, pair the person responsible with a reviewer, so that they may together decide on a joint solution of an issue. Accept the issue, learn from it, and solve it - creating a problem is a learning opportunity.

II. Dependencies - Explicitly declare and isolate dependencies

Go includes a packaging system for installing packages, `go get`. While you can install packages your application uses this way, you have no control over which package will be installed when you run it. As you want to have a repeatable build process, using `go get` is not advised, but you should look to vendoring to find an alternative.

Vendoring

For you guys coming from PHP, Python or Node: there you have `composer`, `pip` and `npm`. For Go, you have a wider choice of package management tools[17] which create a vendor/ folder. I personally recommend using gvt[18], other people also suggested glide[19].

Fetching a dependency with gvt

```
gvt fetch github.com/namsral/flag
```

Using `gvt fetch` is the equivalent of using `go get`, with the difference that the resulting package is written in the `vendor/` folder. You should commit this folder to your VCS, keeping it there for future builds. If you need to update the dependencies for some reason, you can at a later time issue `gvt update`, and then commit the changes to the vendor/ folder.

You can list your current dependencies by issuing `gvt list`.

```
$ gvt list
github.com/namsral/flag https://github.com/namsral/flag master 881a43080604bcf99\
ab1118a814d1cb2c268fc36
```

If you don't want to keep the packages committed to your source repository, you can add this `.gitignore` rule:

[17] https://github.com/golang/go/wiki/PackageManagementTools
[18] https://github.com/FiloSottile/gvt
[19] https://github.com/Masterminds/glide

```
vendor/**
!vendor/manifest
```

This will allow you only to commit the manifest file to your git repository, without any other sources. When you want to make a clean build of your application, you would check out your sources and issue `gvt restore` to fetch all the dependencies declared in the manifest file. You can also use `gvt update` to stay at the tip of your dependencies.

> Note: when using this approach, `gvt` is required in the build environment. Your build may fail because of network connectivity issues, renaming or deleting any of the upstream packages. If you commit your complete vendor folder, it guarantees that you will be able to reproduce the build at any later time.

Caveats about vendoring

Vendoring is an advocated way to provide packages to your application, with more fine grained control as to what version or branch of the package you would like to use. That being said, there are a few points in the small print which you need to consider.

Dependencies

Your code, subpackages or vendored packages might depend on different versions of the same package. This usually means that you'll need to fix some code, so that they can work on the same version, or that you will need to provide some kind of additional versioning for your own packages.

A project which aims to resolve a part of this problem is gopkg.in[20]. When you're importing dependencies, you can specify a tag or branch by constructing an URL which includes this information. There are two url patterns supported.

```
gopkg.in/pkg.v3      â†’ github.com/go-pkg/pkg (branch/tag v3, v3.N, or v3.N.M)
gopkg.in/user/pkg.v3 â†’ github.com/user/pkg   (branch/tag v3, v3.N, or v3.N.M)
```

Unfortunately, your dependency needs to provide branches/tags to support this. Some of them do. If yours doesn't, you might have to fork the original repository, and add some tags or branches, so you can separate the versions.

You can also leverage `gvt` to do this, but you might have to edit the manifest file yourself, as this is not a common use-case. When you do it, keep in mind that `gvt update` will update all packages to the latest revision - this might be problematic, if you're bound to a specific tag or commit.

Relative imports

In your application, you may use something like this:

[20] http://labix.org/gopkg.in

```
1 import "./subpackage"
```

While this is a perfectly valid pattern, it doesn't work with vendoring. If your package is using any kind of vendored imports, your application will not build. If your application lives in `$GOPATH/src/app` as it should, use this import instead:

```
1 import "app/subpackage"
```

This import searches for files in the `$GOPATH/src/app/subpackage` folder, which is exactly where your files are. The reason why you should do this is that relative imports are currently not working correctly with vendoring. People are suggesting to avoid relative imports altogether, but I suspect this may be fixed in a future version. It makes perfect sense for relative paths under your project folder, but not above (with `../`).

> You could use tooling to rewrite parts of your source to change patterns like "./" to "app/", but you'd have to use it instead of `go run` and `go build` as a pre-processing step. While this could be acceptable in your build environment, it would be a hassle to do it in your development environment. My suggestion is to stick to the default tools here - there's no benefit in additional tooling which you have to develop and maintain, unless it provides strong value.

System dependencies

When building docker images, you use the Dockerfile to declare your dependencies. A very minimal Dockerfile for creating a `bash` docker image would look like this:

```
1  FROM alpine
2  RUN apk add --no-cache bash
```

This serves as a *declaration* of a dependency for your application. By building a docker image, the image itself is *isolated* from other containers. The equivalent image on a Debian system might look something like this.

```
1  FROM debian
2  RUN apt-get update && apt-get install bash
```

Ah, it sounds funny, but the RUN command isn't actually needed, as Debian already comes packaged with `bash`. It would theoretically update bash to the latest version, if you're not updating the FROM source image.

When it comes to building docker images, you can compose your images this way. For example, there's an official package for `golang:1.8-alpine`, which you would use for `go run` and `go build` commands. If you wanted to create a "gvt" image, you would create a new Dockerfile, and add on something like:

```
1  FROM golang:1.8-alpine
2  RUN go get -u github.com/FiloSottile/gvt
```

You would repeat this, for every tool which is installed with `go get`, creating a new image for each of them. Or you could just declare one "big" Go image like this:

```
1  FROM golang:1.8-alpine
2
3  MAINTAINER Tit Petric <black@scene-si.org>
4
5  ## install needed packages
6  RUN  apk --no-cache add gcc musl-dev git
7
8  ## add external get packages
9  RUN  go get -u github.com/FiloSottile/gvt && \
10   go get -u github.com/Masterminds/glide && \
11   go get -u github.com/kisielk/errcheck && \
12   go get -u github.com/golang/lint/golint  && \
13   go get -u github.com/goreleaser/goreleaser
```

This is already considered bad practice, because you don't have explicit control over individual applications. If you want to have `gvt`, you should build an image which provides only gvt. This image would be based in a parent golang image.

I don't agree on this point very strongly - the single responsibility principles with building docker images to best practices are dealing with running parts. With such an image, the only running part is the CLI program which you run - go, gvt, glide, or any other from the list.

The problem of violating single responsibility principles comes where you have for example, supervisord, nginx, and php-fpm running in the same container. You can't change php-fpm without modifying all three. The idea is that you only modify one, and this means less possibility of human error (or just isolated to one service, instead of all of them).

> I'm going to play devils advocate here and say that exposing php on a FastCGI interface (TCP) is a bit of a long-stretch. It would depend on the knowledge of the person operating it, but HTTP in comparison is a better understood protocol where people have been known to craft their simple queries (i.e. `GET / HTTP/1.0`) to inspect that something works. FastCGI interfaces are hidden away (on either unix sockets or TCP) and don't provide introspection tools - like the Developer console you're using in your favourite browser. There are of course valid reasons why these services *should* be separated, but alas, they end up together many times for convenience reasons.

Depending on your build/release/run stages, you will most likely separate your dependencies on those. For example, in your build stage you may need two different versions of Go, you may use the official golang[21] image with the respective version tag (`1.8` currently). But for your release/run stages, the `golang` dependency is not required, as you're already dealing with a compiled application binary.

You may have similar development-only requirements for other software, like `npm`, `pip`, `composer`, `yarn`, `webpack` and many others. Include what you like, but choose carefully.

In some cases the simplest way to add a dependency is to issue a `git clone`, while in other cases you might be better off using what is available on Docker Hub (or Alpine/Debian/... package mirrors). If you add all your dependencies to your application development Dockerfile, it will inevitably take a very long time to build.

When we will talk about backing services, you should know that there are a number of "official" docker images available on the Docker Hub[22] and you may choose between them to use, for example `percona:5.5`, `percona:5.7`, or extend or build any image that fits your requirements.

It's not uncommon for people to build their packages from sources, but it does take additional effort to keep the size of such builds in check. Versioned docker images also serve not only as a declaration of dependency, but also as a possible way to roll back your releases (more about that in Chapter V.).

[21] https://hub.docker.com/_/golang/
[22] https://hub.docker.com/explore/

When creating your Dockerfiles, please follow the single responsibility principle[23]. If your docker image is self-contained, migration for it between hosts is easily accomplished. When your docker image contains multiple services, you might have to deal with several graceful shutdowns to effectively migrate data, which will introduce operational complexity and risks. If you do this, you are also clearly violating several chapters of 12FA, most notably dependency isolation from this chapter, and chapter IX. Disposability.

Depending on your development patterns, you should pay attention to provide small Docker images. I tend to travel and work from countries with poor or limited internet connectivity, where every megabyte saved can improve my productivity drastically. I vividly remember downloading images about 100MB in size for about 15 minutes. If you can make a coffee and drink it before the image finishes downloading, that's a good hint that you should optimize it.

[23] https://en.wikipedia.org/wiki/Single_responsibility_principle

III. Config - Store config in the environment

One of the most important 12 Factor Application principles is dictating how your configuration should be declared and passed to your application. If you're using a database instance, the location and credentials for that database instance should come from the environment where your application is run. This means that it should be passed via an environment variable, via a command line argument, or even via a configuration file which must be passed as a command line argument.

The "core" of this requirement of a 12 factor app is to remove experienced bad practice. People defining constants in the code have disabled themselves from changing the environment without rebuilding the whole application. They may have committed database credentials into source control - even with policies regarding database usernames and passwords, this data may be sensitive in other cases, as for example, exposing AWS credentials. Even keeping a configuration file with such information is sensitive - it may be added to source control by mistake, causing the same problem.

All configuration for environment needs to be managed outside of your application. Imagine a basic requirement that your application can be open sourced or the source code may just be bought by a third party. You don't want to expose your AWS or other sensitive credentials. It's always worth to ask yourself: can I publish my application code to the public, without getting burned?

It's a rhetorical question: people unknowingly published their AWS credentials and woke up to find out that their Amazon bill is in the thousands. It's the story of Ryan Hellyer[24], a WordPress developer who managed to leak his credentials to GitHub and ended up with a $6K bill, another got a $50K bill[25] and such stories are not uncommon. Amazon might waive some fees here, but having best practices in place that avoid your exposure is priceless.

[24] https://wptavern.com/ryan-hellyers-aws-nightmare-leaked-access-keys-result-in-a-6000-bill-overnight

[25] https://www.quora.com/My-AWS-account-was-hacked-and-I-have-a-50-000-bill-how-can-I-reduce-the-amount-I-need-to-pay

Flags

When it comes to the standard lib, there's the flag[26] package, which implements command-line flag parsing. The third party extension to this is the namsral/flag[27] package, which is a drop in replacement for the standard lib package with the addition that it parses files and environment variables.

The namsral/flags example

```
1  package main
2
3  import (
4    "fmt"
5    "github.com/namsral/flag"
6  )
7
8  func main() {
9    var age int
10
11   flag.IntVar(&age, "age", 0, "age of gopher")
12   flag.Parse()
13
14   fmt.Print("age:", age)
15 }
```

The flags example shows just how easy it is to pass ENV variables and use them inside Go programs. Using docker, you can pass the ENV variables with the `-e` option, like this:

Passing ENV variables with docker

```
1 #!/bin/bash
2 docker run --rm -e "AGE=35" -v $(pwd):/go/src/app -w /go/src/app golang go run f\
3 lags.go
```

Passing config to packages

With the flag package there's the option that individual packages will define what kind of flags they need. This might become a bit of a problem - after all, you can't prevent that two different packages will need the same configuration by name. What you need to enforce is where you will define your flags and how you will pass them to the correct location where they are needed.

[26] https://golang.org/pkg/flag/
[27] https://github.com/namsral/flag

The suggested practice is to declare flags in the main package, where they are also injected into packages as needed. As the flags are defined in one place, there is less chance with conflicts with other packages. Some public packages of course define their own flags - as such, I'd recommend filing issues against the offending packages to provide an idiomatic way for setting configuration flags via some kind of functional options API.

Best practices

I would recommend that you should define your flags in the `func main()` block. By doing this, they become unavailable for reading/writing by other functions from the global scope, and you have full control how this configuration is passed to packages and other functions. The side effect of this is also that packages have a declared interface over which configuration is set.

My current approach is to provide functional options to packages, as they require it. For example, the `redis.so` from my `services` package may look like this:

Functional options with Redis

```
1  package services
2
3  import "time"
4  import "github.com/garyburd/redigo/redis"
5
6  // Redis connection struct
7  type Redis struct {
8    conn                                    redis.Conn
9    address                                 string
10   connectTimeout, readTimeout, writeTimeout time.Duration
11 }
12
13 type RedisOption func(*Redis)
14
15 // RedisAddress - specify hostname of redis server (including port)
16 func RedisAddress(address string) RedisOption {
17   return func(do *Redis) {
18     do.address = address
19   }
20 }
21
22 // RedisConnectTimeout - set connection timeout for the Redis client
23 func RedisConnectTimeout(timeout time.Duration) RedisOption {
24   return func(do *Redis) {
25     do.connectTimeout = timeout
26   }
```

```
27  }
28
29  // RedisReadTimeout - set a read timeout for a Redis operation
30  func RedisReadTimeout(timeout time.Duration) RedisOption {
31    return func(do *Redis) {
32      do.readTimeout = timeout
33    }
34  }
35
36  // RedisWriteTimeout - set a write timeout for a Redis operation
37  func RedisWriteTimeout(timeout time.Duration) RedisOption {
38    return func(do *Redis) {
39      do.writeTimeout = timeout
40    }
41  }
42
43  // NewRedis creates a new redis connection
44  func NewRedis(options ...RedisOption) *Redis {
45    redis := &Redis{
46      address:        "redis:6379",
47      connectTimeout: time.Second,
48      readTimeout:    time.Second,
49      writeTimeout:   time.Second,
50    }
51    for _, option := range options {
52      option(redis)
53    }
54    return redis
55  }
56
57  var (
58    redisConnections = make(map[string]*Redis, 0)
59  )
60
61  // Save redis connection for reuse
62  func (t *Redis) Save(names ...string) {
63    connection := "default"
64    for _, name := range names {
65      connection = name
66    }
67    redisConnections[connection] = t
68  }
```

```
69
70  // GetRedis retrieves a saves redis connection for reuse
71  func GetRedis(names ...string) (redis.Conn, error) {
72    var err error
73    connection := "default"
74    for _, name := range names {
75      connection = name
76    }
77    r, ok := redisConnections[connection]
78    if !ok {
79      r = NewRedis()
80      r.Save(connection)
81    }
82    if r.conn == nil {
83      connectTimeout := redis.DialConnectTimeout(r.connectTimeout)
84      readTimeout := redis.DialReadTimeout(r.readTimeout)
85      writeTimeout := redis.DialWriteTimeout(r.writeTimeout)
86      r.conn, err = redis.Dial("tcp", r.address, connectTimeout, readTimeout, writ\
87  eTimeout)
88    }
89    return r.conn, err
90  }
```

The functions that return a `RedisOption` struct are passed as variadic arguments to the `NewRedis` constructor function. The constructor takes care of reasonable defaults, while honouring passed RedisOption structs to allow customisation of all options.

> This way worked best for me, but your examples may be less complex. In such cases, passing whatever you need as function arguments might suffice for your case. If you look at main.go you can see that the `port` flag is used without any special interfaces.

godotenv

GoDotEnv[28] is a port of the `dotenv` package available with Ruby. This package loads ENV variables from a `.env` file which should be located in the root of your project. Keep in mind, that this file shouldn't be committed in your git repository, but should come to your development, staging or production environments from your infrastructure management solution or simply as part of your testing or deployment environments.

To use the package, `import github.com/joho/godotenv` and add the following line to the beginning of your main function. It should be added before using the `namsral/flag` package:

```
1 godotenv.Load()
```

We can use the flags example from the start of this chapter and set the required configuration flag into the `.env` file which we create.

```
1 echo AGE=1337 > .env
```

Environment variables have precedence over the variables declared in .env; This enables you to provide sane defaults via the `.env` file and then provide higher-level overrides via declared environment variables. This means if you will provide `-e AGE=35` with the docker command line, it will ignore the value written in your `.env` file.

[28] https://github.com/joho/godotenv

Viper

Viper is a popular configuration library that's designed with 12 factor applications in mind.

> Viper is a complete configuration solution for go applications including 12 factor apps. It is designed to work within an application, and can handle all types of configuration needs and formats. Viper can be thought of as a registry for all of your applications configuration needs.

You can create a viper instance by calling `viper.New()`. This allows you to manage configuration for multiple parts of a large application. If you don't need it, you may use `viper` directly. Let's create an utility function which will aid us to load configs and set default values.

```
1  func readConfig(filename string, defaults map[string]interface{}) (*viper.Viper,\
2   error) {
3    v := viper.New()
4    for key, value := range defaults {
5      v.SetDefault(key, value)
6    }
7    v.SetConfigName(filename)
8    v.AddConfigPath(".")
9    v.AutomaticEnv()
10   err := v.ReadInConfig()
11   return v, err
12 }
```

Viper supports nested structures in the configuration, this is why the defaults are of the type `map[string]interface{}`. We loop through them and set default values for individual keys. Viper will use these after it checks that the value is not defined either in the configuration or the environment.

By calling `SetConfigName` we set the base filename where the config will be read from. Viper will search for supported formats by trying to open `[filename].json`, `[filename].yaml` and other supported formats. Viper will never read from `[filename]` directly, but will always append the extension for supported file types.

As we want to read the configuration from the current directory, we call `AddConfigPath()` with the parameter `.` (current directory). If you wanted to load configuration from multiple directories, you could repeat this call with different paths.

Finally, we call `AutomaticEnv` to ensure that Viper will read from environment variables as well. With this, we have everything we need to read any configuration format that Viper supports.

```
func main() {
  v1, err := readConfig(".env", map[string]interface{}{
    "port":     9090,
    "hostname": "localhost",
    "auth": map[string]string{
      "username": "titpetric",
      "password": "12fa",
    },
  })
  if err != nil {
    panic(fmt.Errorf("Error when reading config: %v\n", err))
  }

  port := v1.GetInt("port")
  hostname := v1.GetString("hostname")
  auth := v1.GetStringMapString("auth")

  fmt.Printf("Reading config for port = %d\n", port)
  fmt.Printf("Reading config for hostname = %s\n", hostname)
  fmt.Printf("Reading config for auth = %#v\n", auth)
}
```

We provide the default values with the second parameter to readConfig. As with the config files that Viper supports, nested structures are supported here as well. If we create a short `.env.yaml` file like this, viper will read from it nicely:

```
auth:
  username: black
  password: notthere
```

Keep in mind, if no config files are present, you will end up with an error. If you have an empty `.env.json` you will end up with an error as the contents are not a valid JSON document, you need at least `{}` to correctly parse JSON.

You can even read nested configuration structures with viper, by delimiting nested keys with `.` (dot). This means you can do `v1.getString("auth.username")` and get the value from the username key within the auth map. However, keep in mind that there's no good way to override nested structures with environment variables at this point. You can do the following:

```
v1.SetEnvKeyReplacer(strings.NewReplacer(".", "_"))
```

And when you would provide the environment variable `AUTH_USERNAME`, you can retrieve it with `v1.getString("auth.username")`. However, if you will retrieve the full `auth` value, the nested value

will not be updated with the value from the environment. As such, it doesn't make much sense to rely on environment variables if you need nested structures. Viper could do a better job of importing those.

The full example is available in the book samples GitHub[29]. There are two scripts provided which run the example with Docker, once with and once without environment variables.

[29] https://github.com/titpetric/books/tree/master/12fa-docker-golang/chapter3/viper

IV. Backing services - Treat backing services as attached resources

Backing services can range from any form of database to NoSQL data stores and proxies. Generally, these are services that are not exposed to the public directly but are only used in your application. It makes sense for them to be reachable on the same network where your application lives, either with a custom overlay network which we defined in the introduction chapter, or with some form of private LAN (Local Access Network).

The main point of why we treat backing services as attached resources is that they may be replaced based on any criteria. If a database is misbehaving, it's possible to upgrade it or migrate it to a different host. If your Redis instance is crashing for some reason, you might migrate to a newer version with bugfixes easily. If you need to scale your service, you might spread it over multiple hosts with docker swarm, or you might completely switch out a single redis instance for multiple ones that can be served behind an instance of twemcached (or nutcracker).

Data resources

A lot of people advocate that database workloads are not suited to docker. I suspect that the argument for this is mainly based on the fact that docker containers are ephemeral, meaning that they and their data can be destroyed with a simple `docker rm` command. When it comes to storing data persistently, we can create what is called "data volumes" - of which there are two types.

Data volume containers

The first and in my experience less common type is to create a data volume container.

> Data volumes are designed to persist data, independent of the container's life cycle. Docker therefore never automatically deletes volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container.

Projects like rancher/convoy[30] approach this use pattern by providing a custom volume filesystem, that provides ways to manage it - namely by providing utilities to create snapshots and enable migration between hosts via some sort of "cloud" storage, S3 being an example of this. In such cases, data is stored in an additional container, which is less prone to deletion when you stop and remove your containers.

This way is suitable (and recommended) when you're dealing with cloud deployments. For example, Google Compute Engine provides what is called a "persistent volume". For all intents and purposes, the management of these storage volumes is separated from the management of containers. This means, that the containers will request these resources without any knowledge of the underlying infrastructure.

Data volumes are created as they are referenced, so when using `-v` to attach a volume, the volume itself will be created if it doesn't exist. A very banal example of creating, writing and reading from such a volume:

```
# docker run --rm -v bash-storage:/app bash bash -c 'echo "hello world" > /app/h\
ello'
# docker run --rm -v bash-storage:/app bash bash -c 'echo $(</app/hello)'
hello world
```

Such volumes can be listed with the `docker volume ls` command:

[30] https://github.com/rancher/convoy

```
# docker volume ls
DRIVER              VOLUME NAME
local               bash-storage
```

Out of the box, docker volume command comes with `create`, `inspect`, `ls` and `rm`. The keen observer will note that there are no available ways to archive (snapshot, backup) provided by docker itself. The official way to copy the data from such a container is to attach the volume to a container, and from there copy the data using your choice of tools (like `tar` and `rsync`) to any local storage or remote location.

For more information, you can consult the official Docker documentation on volumes[31].

Data volume

The second and more common option in on premises deployments is to create volume mounts of a host directory to store it's data. When you stop the container, the data is available as is on the host, where it can be archived or copied to another machine. Stopping the container might not be needed, this depends on the container you are running and how it is storing data. For example: if you have a web server, which stores file in a writeable upload folder - these files are safe to copy without turning off the server. If you have a MySQL instance, you may use Percona Xtrabackup[32] to create a snapshot of your database data while the container is running.

Volume mounts by itself may be used not just for data storage, but also for your application itself. In case of PHP applications, you may use a `php` container where you pass your scripts into the container, and may even update them while the container is running. If you're writing your application in Go - as it is compiled, you will be better off creating a new container for each one of your application builds. You can't replace a running binary without restarting a container, and if you're restarting a container you need to think about how you will start a new container and stop the old one without interrupting the service you have.

If you're dealing with "rolling update" scenarios, like it's common with Go applications, a good way to do that is with using a Docker swarm cluster. For more information, please consult the rolling update chapter of docker swarm documentation[33] and the section dedicated to docker swarm in Chapter 8.

[31] https://docs.docker.com/engine/tutorials/dockervolumes/#/backup-restore-or-migrate-data-volumes

[32] https://www.percona.com/software/mysql-database/percona-xtrabackup

[33] https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/

Example backing services

Depending on your application, you may need any of the few, very common, backing services which I'm giving you as an example here. There are some tricks to running them in production, most notably a good comprehension of `sysctl` - open file limits and networking tweaks.

Please check the samples for more examples of not only starting such backing services, but how to interface with them when they are running. Generally they provide one service and in addition to that they also provide some tooling to interface with it. For example, MySQL provides the command line client `mysql`, database data dump utility `mysqldump` and in our case, the complete Percona toolkit, which is a good thing to get familiar with. In the same way, Redis provides `redis-cli`, a command line client where you can issue manual commands against it, without resorting to telnet or a custom-built client.

MySQL

For performance reasons I'm a proponent of the Percona builds of MySQL. The build was a life saver years ago when we were hitting performance issues with InnoDB tables on a vanilla MySQL server. Back then, before docker came along - mitigation strategies in such conditions were few - especially if you didn't have faster hardware on hand. Obviously, what's most important is to have *time* to investigate and fix such issues, and Percona also provides a very useful instrumentation layer that allows you to diagnose usage patterns and issues with incorrectly set indexes and possibly determine a cause that will allow you to make effective changes to your application and keep the database layer running smoothly. Percona also provides a very valuable resource with their MySQL Performance Blog[34] which is a must-read for any back-end developer and MySQL DBA.

Starting a MySQL container

```
#!/bin/bash
#
# services/mysql/bin/run -> MYPATH= services/mysql, NAME= mysql
#
MYPATH=$(dirname $(dirname $(readlink -f $0)))
NAME=$(basename $MYPATH)
if [ ! -d "$MYPATH/data" ]; then
    mkdir -p $MYPATH/data
fi

DOCKER_IMAGE="percona:latest"

#
```

[34] https://www.percona.com/blog/

```
# you can use this image if you want
# percona packaged with xtrabackup
#
#DOCKER_IMAGE="titpetric/percona-xtrabackup"

#
# generate root password to file for convenience
#
PASSWORD="default"
perl -pi -e "s/^password.+$/password=$PASSWORD/g" ../conf/.my.cnf

if [ ! -d "$MYPATH/data" ]; then
    mkdir -p $MYPATH/data
fi

# stop and remove existing container and start a new one

docker stop $NAME
docker rm $NAME
docker run --restart=always \
    -h $NAME \
    --name $NAME \
    --net=party \
    -v /src/$NAME/backups:/backups \
    -v /src/$NAME/conf/conf.d:/etc/mysql/conf.d \
    -v /src/$NAME/data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=$PASSWORD \
    -d $DOCKER_IMAGE
docker cp /src/$NAME/conf/.my.cnf $NAME:/root/
```

Connecting and using MySQL

Connecting to and using a MySQL instance is very straightforward. Depending on the size and fragmentation of your service, you might want to use different connections at the same time. My client of choice is `jmoiron/sqlx`, which you should vendor and import. We'll be using a subpackage `service` so provide us with a database factory.

```
package service

import (
  "strings"
  // import database driver for mysql
  _ "github.com/go-sql-driver/mysql"
  "github.com/jmoiron/sqlx"
)
```

For example, a multi-tenant SaaS app might use at least two databases - one created for the client, and a central management database which might have read-only subscription details, like billing and invoices and similar. The app should use two distinct database connections in this case.

When you're using multiple connections, you also have to think about configuring them. There has to be a way to get the configuration from, basically, your main function, to all the components that might use one or more databases.

For this purpose, I created a struct with the following interface:

```
// Database connection factory object
type Database struct {
  // hold database config
  dsn *string
}
```

Getting a database connection is done with the method `Get`. We don't need to close the connection as sqlx maintains an internal database connection pool. I'm adding some default configuration inside the Get method, to set a reasonable character set and parse time fields into Go native structures.

```
// Get creates and returns a new database connection
func (r *Database) Get() (*sqlx.DB, error) {
  dsn := *r.dsn
  if !strings.Contains(dsn, "?") {
    dsn = dsn + "?"
  }
  // set some default values for dsn config
  defaults := map[string]string{
    "collation": "utf8_general_ci",
    "parseTime": "true",
    "loc":       "Local",
  }
  // append default values
  for key, value := range defaults {
```

```
15      if !strings.Contains(dsn, key+"=") {
16        dsn = dsn + "&" + key + "=" + value
17      }
18    }
19    dsn = strings.Replace(dsn, "?&", "?", 1)
20    return sqlx.Open("mysql", dsn)
21  }
```

The interface might be improved depending on how close you want your usage to be to `sqlx.DB`. As we were saying before, configuration needs to be set per-connection factory, so we can have multiple connections. I'm relying on the `flag` package in Go standard library, but it could be replaced by any which satisfies the interface here. Using `namsral/flag` is a drop-in replacement, as well as `spf13/pflag`.

```
1  // Interface for used flag functions
2  type databaseFlags interface {
3    String(name, value, usage string) *string
4  }
5
6  // Flags registers a new parameter in a compatible flags package
7  func (r *Database) Flags(name, usage string, flag databaseFlags) {
8    r.dsn = flag.String(name, "api:api@tcp(db1:3306)/api", usage)
9  }
```

Creating a database factory and configuration:

```
1  flags := flag.NewFlagSet("default", flag.ContinueOnError)
2  database := &service.Database{}
3  database.Flags("dsn", "DSN for database connection", flags)
4  flags.Parse(os.Args[1:])
```

Since sqlx maintains an internal database connection pool, we can create only one `*sqlx.DB` object and use it as a singleton instance. For example, using the database class to list available databases becomes very simple:

```
1  func listDatabases(db *sqlx.DB) ([]databaseName, error) {
2    result := []databaseName{}
3    err := db.Select(&result, "show databases")
4    return result, err
5  }
```

This should give you the basics for using a database connection. Depending on your usage patterns, you will decide on the structure which best works for you. When you're using other clients like Redis, you might need to create instances for every request, or you may choose to create a connection pool much like sqlx does internally.

Redis

When it comes to NoSQL stores, Redis might be my favourite by far. The author approaches it more than just a pet/hobby project, and the project earned credible sponsors like VmWare along the way. If for nothing else - consider the comparison with Memcached and Redis: Memcached is a simple key-value store, which doesn't have any kind of storage back-end, any way to list the keys which are stored inside it, and no means of replication to migrate what might be a large data set from one machine to the next. Redis has disk-based storage ("rdb") which you can snapshot and replicate. The cherry on top are of course the added data types which make Redis a true data store, and not just a key-value cache.

Running a single instance with performance settings is easy.

Starting a Redis container

```
#!/bin/bash
MYPATH=$(dirname $(dirname $(readlink -f $0)))
NAME=$(basename $MYPATH)
DOCKER_IMAGE=$(<$(dirname $(dirname $(readlink -f $0)))/docker.image)

# remove redis warnings
sysctl vm.overcommit_memory=1
echo never > /sys/kernel/mm/transparent_hugepage/enabled

SYSCTL="--sysctl net.core.somaxconn=4096"

echo -n "$NAME: "
docker stop redis && docker rm redis
docker run --restart=always -h $NAME --name $NAME $SYSCTL --net=party -d $DOCKER\
_IMAGE </dev/null
```

Similarly to MySQL, the client library which we use for Redis, garyburd/redigo[35], provides a connection pool struct[36]. Unlike MySQL Redis is single threaded, so commands against a single server will run without parallelism.

> For instance a bartender is able to look after several customers while he can only prepare one beverage at a time. So he can provide concurrency without parallelism.

This means that we will need to connect our Redis pool to several servers, to achieve parallelism up to the number of server we connect to. There's a command in Redis which illustrates this mode

[35] https://github.com/garyburd/redigo

[36] https://godoc.org/github.com/garyburd/redigo/redis#Pool

of operation: `DEBUG SLEEP [seconds, float]`. If we run `DEBUG SLEEP 0.1` and immediately after `DEBUG SLEEP 0.2`, both commands will take about 0.3 seconds to complete on one Redis server. We can do this from two goroutines and observe this result. Even if Go provides concurrency and parallelism with goroutines, one single-threaded Redis server will be a limiting factor. At least running multiple instances is very easy with Docker.

There's not a lot of complexity in comparison with the example for MySQL. As we might need to define multiple servers, I decided to extend the flags interface, so we can call it multiple times in order to declare configuration for multiple Redis servers:

```
type Redis struct {
  pool *redis.Pool
  dsn []*string
  serverIndex int
}

// Interface for used flag functions
type redisFlags interface {
  String(name, value, usage string) *string
}

// Flags registers a new parameter in a compatible flags package
func (r *Redis) Flags(name, value, usage string, flag redisFlags) {
  r.dsn = append(r.dsn, flag.String(name, value, usage))
}
```

As such, we can call the `Flags` method multiple times, adding additional server configurations which will then be used by the Redigo Pool structure:

```
// Get a single redis server name (based on flags registrations)
func (r *Redis) getServerName() string {
  name := *r.dsn[r.serverIndex]
  r.serverIndex++
  if r.serverIndex >= len(r.dsn) {
    r.serverIndex = 0
  }
  return name
}

// Returns a connection which needs to be closed
func (r *Redis) Get() redis.Conn {
  if r.pool == nil {
    r.pool = &redis.Pool {
```

```
      MaxIdle: 3,
      IdleTimeout: 240 * time.Second,
      Dial: func() (redis.Conn, error) {
        return redis.Dial("tcp", r.getServerName())
      },
    };
  }
  return r.pool.Get()
}
```

Each connection that will be returned by `service.Redis{}.Get` needs to be closed by calling the `Close` method on that object. But since it's trivial to provide our own utility method to issue commands against a Redis connection, I added a `Do` method to do just that:

```
// Execute a command against a temporary pool connection
func (r *Redis) Do(commandName string, params ...interface{}) (interface{}, erro\
r) {
  conn := r.Get()
  defer conn.Close()
  return conn.Do(commandName, params...)
}
```

With the above we can declare as many servers as we want with any flag compatible package, and use a simple utility method to issue commands against a Redis instance.

```
// set up redis service
redis := &service.Redis{}
redis.Flags("redis1", "redis1:6379", "Redis DNS", flags)
redis.Flags("redis2", "redis2:6379", "Redis DNS", flags)
redis.Flags("redis3", "redis3:6379", "Redis DNS", flags)
```

As each call to `redis.Do` is blocking due to the single-threaded nature of Redis, we need to issue the calls in goroutines in order to achieve parallelism. We can leverage Go channels to wait for the result of each goroutine.

```
fmt.Printf("[%.4f] Start\n", service.Now())

sleep1_chan := make(chan string, 1)
sleep2_chan := make(chan string, 1)

go func() {
  conn := redis.Get()
  defer conn.Close()

  fmt.Printf("[%.4f] Run sleep 100ms\n", service.Now())
  sleep1, err := conn.Do("DEBUG", "SLEEP", "0.1")
  if err != nil {
    sleep1 = "ERROR"
  }
  sleep1_chan <- sleep1.(string)
}()

go func() {
  conn := redis.Get()
  defer conn.Close()

  fmt.Printf("[%.4f] Run sleep 200ms\n", service.Now())
  sleep2, err := conn.Do("DEBUG", "SLEEP", "0.2")
  if err != nil {
    sleep2 = "ERROR"
  }
  sleep2_chan <- sleep2.(string)
}()

var result string
result = <-sleep1_chan
fmt.Printf("[%.4f] End Sleep 100ms, result %s\n", service.Now(), result)
result = <-sleep2_chan
fmt.Printf("[%.4f] End Sleep 200ms, result %s\n", service.Now(), result)
```

There's some overhead in creating a TCP connection to Redis, but as you can see, the commands are issued against different servers, effectively achieving parallelism:

```
[0.0000] Start
[14.5576] Run sleep 100ms
[14.6821] Run sleep 200ms
[114.8777] End Sleep 100ms, result OK
[214.9519] End Sleep 200ms, result OK
```

Docker registry

The Docker registry is a container designed to provide docker image storage, much like Docker hub provides it for free for the public. To have private docker image storage available, you can set up your own registry server, much like you can run MySQL, Redis or anything else.

Starting a Registry server

```
#!/bin/bash
NAME="registry"
DOCKER_IMAGE="registry:2"
# by default storage is in ../storage
STORAGE=$(dirname $(dirname $(readlink -f $0)))/storage
if [ ! -d "$STORAGE" ]; then
        mkdir $STORAGE
fi
# stop and remove existing instance
docker stop $NAME && docker rm $NAME
docker run -d -p 5000:5000 --restart=always --name $NAME -v $STORAGE:/var/lib/re\
gistry $DOCKER_IMAGE
```

When you'll start your own registry, connecting to it via SSL is the default option. This means, that a fully qualified SSL certificate should be set up, and a proxy which forwards the domain to your registry server. The great LetsEncrypt CA[37] can be used to generate SSL certificates for any domains you might have.

Logging into a registry

Logging into the Docker hub is needed if you would like to push your images to your username on Docker hub.

[37] https://letsencrypt.org

```
# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't \
have a Docker ID, head over to https://hub.docker.com to create one.
Username: titpetric
Password:
Login Succeeded
```

After you have made a successful login, you are able to "push" your images to your Docker Hub account. For example, I can push my docker image from a later chapter to Docker Hub:

```
# docker push titpetric/gotwitter
The push refers to a repository [docker.io/titpetric/gotwitter]
1592f0386889: Pushed
c32cc9520ad2: Pushed
60ab55d3379d: Pushed
latest: digest: sha256:e0a7ad8d994273710d92c92fdc3c30244b13fb519638bfefb84da1aae\
616bec1 size: 949
```

When you'll move on from Docker Hub to hosting your own private registry, you'll notice a few additional benefits:

1. using your own domain (registry.example.com for example),
2. there are no non-readable IDs involved (AWS account id, regions, etc.)
3. consistent URLs for images and mnemonic devices to type them out

```
# docker login registry.example.com
Username:
Password:
...
# docker push registry.example.com/gotwitter
The push refers to a repository [registry.example.com/gotwitter]
...
```

After you login to your registry, all the images you push are also available for pulling everywhere you are logged in. This means that you could just use `git pull registry.example.com/gotwitter` on each of your docker hosts or docker swarm, and the image would pull any updates to your hosts. This is also a deployment step when you're publishing your application to anywhere.

> Hosting your own private Docker registry can be a bit confusing, as there is no interface provided currently for it. Basically, what it does is expose a full API to the docker command line client, but it doesn't give you a GUI or a simple way that would let you - for example - delete an image:tag that you pushed to it. The project kwk/docker-registry-frontend[a] might solve this issue.
>
> [a] https://github.com/kwk/docker-registry-frontend

Private docker registry as a service

In case you're like me and would like to farm out the management of a Docker registry, there are many services available. I'll concentrate on the main three that provide it:

1. Docker Hub[38],
2. Amazon EC2 Container Registry (Amazon ECR)[39],
3. Google Container Registry (GCR)[40]
4. Quay.io[41]

The Docker Hub service provides private docker image storage as well. After signing up, you get one free private docker image, and beyond that you have a progressive subscription plan. Docker hub also provides a CI platform where you can create automated builds for your Docker images (Dockerfile). Generally, the way Go applications are built however, this might be a redundant feature for you, and you might look to exclusively storage options like ECR or GCR.

Amazon ECR service[42] is available via your AWS console. Amazon provides an AWS CLI tool[43] that provides common functionality which you need when interfacing with their docker registry. The setup is very fast, however the images which are stored in the container registry are referenced by a very unfriendly "aws_account_id.dkr.ecr.us-east-1.amazonaws.com/image".

Using Google container registry is quite simple, you can start on a free plan (they give you $300 credit to use anywhere in Google Cloud) and then charge you on used storage and used egress traffic. Google provides the Google Cloud SDK[44] set of tools that make interacting with their services, including GCR, a bit easier (much like AWS CLI). The main difference over vanilla docker commands is that Google provides different modes of authentication, most notably access tokens - for more information you may review their advanced authentication methods page[45]. The images are

[38] https://hub.docker.com/
[39] http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html#use-ecr
[40] https://cloud.google.com/container-registry/
[41] https://quay.io/
[42] https://aws.amazon.com/ecr/
[43] http://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR_AWSCLI.html
[44] https://cloud.google.com/sdk/docs/
[45] https://cloud.google.com/container-registry/docs/advanced-authentication

referenced with `gcr.io/my-project/my-image` which gives a more friendly feeling in comparison with Amazon ECR.

Both options are unfriendly to a point, not providing access via vanilla docker commands, but wrapping them in their own cli/sdk scripts respectively. While this works fine in terms of using their own products for deployment of your applications, you might do well to look to Quay.io as the better option - it's offering is very similar to that of Docker Hub, but it does provide some technological advancements and a full interface for when you want to host the registry yourself. For example with the official docker registry image, there is the caveat of how you would go about deleting images which have been stored in it (outdated builds, etc.). Quay currently bridges this gap by providing a full administration interface for your registry.

Managing your hosts with Portainer

If you are more familiar with having some kind of front-end interface to your infrastructure, you might appreciate running [Portainer]. Running portainer is very simple:

```
# docker run -d -p 9000:9000 \
      -v /var/run/docker.sock:/var/run/docker.sock \
      portainer/portainer
```

By passing the docker.sock file to portainer we already enable managing a local docker instance. You can visit your host IP and port 9000, and Portainer should open. It will ask you to define a password for the admin account, ask which Docker host you'd like to connect to, and that's it.

You have your whole Docker host or swarm available over a friendly GUI interface. Portainer includes very helpful App Templates, which you can use to create some of your common services. For example, you can create an S3 API compatible service called Minio, by only configuring some common options like which volumes you would like for storage. It's awesome, and you should look at it.

V. Build, release, run - Strictly separate build and run stages

The 12 Factor Application guidelines advocate separating your software development process into three cycles:

- `build` - where you prepare your code into a bundle (executable or docker image),
- `release` - where you release this bundle (release to GitHub or push docker image),
- `run` - the final stage where you can take any release and run it

The separation between these stages ensures that development takes place in the correct environment for it - if we would be dealing with a scripting language like Node or PHP, there would be an opportunity to modify it in the runtime environment but it would add complications like how to propagate these changes back to the build stage. By creating releases of your software, you also enable rolling back any changes that might disrupt your production environment.

As Go is a programming language with a compiler - the "build" stage takes care of translating source code into machine-code executables, and the "release" stage publishes these executables for example, on GitHub, or as a docker container on Docker Hub.

Building Go applications

With Go, we can build executables for multiple architectures and platforms.

For example, the architectures are:

- `amd64` - a 64bit x86 architecture,
- `386` - a 32bit x86 architecture,
- `arm` - an ARM architecture (Raspberry PI for example)

When building your application, you can pass `GOARCH` with the target platform. Similarly, you can pass `GOOS` with the target operating system: `linux`, `windows`, `freebsd`, `darwin` and others are supported. This means that you can package your application for different platforms, as long as you can build the executable for them.

When your application is compiled, it uses the libraries that are available in your system. This process is called dynamic linking[46]. When you transfer your executable from one system to another (or from one Docker image to another), it will likely have a different set of libraries available, causing the application not to work.

When building docker images you should use `golang:1.8-alpine` for example to build your image, and then use the underlying base image `alpine:3.5` to run the resulting binaries. If you use any other Docker image for golang, you can inspect the Docker Hub page[47] for the Dockerfiles, to see which base image is safe to use for your application.

This is a typical way to check what kind of executable you have:

```
# file ./gotwitter
./gotwitter: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically li\
nked, interpreter /lib/ld-musl-x86_64.so.1, not stripped
```

Another (better) approach is called static linking[48] where the resulting application embeds the libraries which are needed. This way, the binary is truly portable and you can move it to other compatible operating systems and platforms, regardless of which libraries they provide.

The way to achieve static linking of Go applications is to pass some options when building your application which will be passed on to the underlying linker. Go build supports the `--ldflags` parameter, which can be used to pass the `-static` option to the underlying compiler and linker. You would also need to pass the environment `CGO_ENABLED=0` to disable the `cgo` tool and use `gcc`, like it would if you would be doing cross-compilation to a different platform/architecture.

[46] https://en.wikipedia.org/wiki/Dynamic_linker
[47] https://hub.docker.com/r/_/golang/
[48] https://en.wikipedia.org/wiki/Static_library

```
# file gotwitter
gotwitter: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linke\
d, not stripped
```

For more information about `CGO_ENABLED`, `GOOS` and `GOARCH`, consult optional environment variables[49] and the cgo tool[50] documentation, if you need some deeper knowledge of the subject.

> If you wanted to save a few MB, you could use the docker `scratch` image. It literally provides nothing, but at the same time is good enough to run your application. If you're doing some HTTP/S API calls and similar, it might be better to stick with Alpine. Having 5MB overhead that gives you a reasonable system with a package manager is priceless.

[49] https://golang.org/doc/install/source#environment
[50] https://golang.org/cmd/cgo/

Building Docker images

When building a Docker image, it's advisable to tag your images with a specific version. This may be either a date/time string, any incremented number or versioning system like semver[51] or the commit ID from your git repository. If you use the commit ID, it means that you would have a way to rebuild the exactly same Docker image based on the source code in that commit.

```
GITVERSION=$(git rev-list HEAD | head -n 1 | cut -c1-6)
```

The above command would set a GITVERSION variable in bash. The way it works it first gets a list of all your commit hashes. With `head -n 1` we extract the latest commit, and with `cut -c1-6` we only keep the first 6 letters of the commit hash. This information is usable also if you want to review your code - with the image tagged, you know exactly at which revision your source code was when used for building the image.

I like to base my builds on the commit ID of whatever it is I'm building. I set up an argument and an environment variable, that will hold this information when the container runs.

```
ARG GITVERSION=development
ENV GITVERSION ${GITVERSION}
```

Generally, what you need to know here is that the `ARG` command provides a variable, which you can pass during `docker build` with the `--build-arg` option. By itself, the value of the ARG parameter doesn't persist into the container. This is why I also use the `ENV` command, to persist the value that is given. When you're running your container, you can inspect it by doing something like this:

```
# docker exec -it monocms-cache-nutcracker env | grep GITVERSION
GITVERSION=85990e
```

With this information, I can checkout the git repository at exactly this commit, perhaps change something in the Dockerfile, and rebuild it exactly as it was (only better?). This same version information may also be used for reverting the build, as long as you know the order of the images, as they were built or deployed to production environments.

[51] http://semver.org/

Continuous integration with Codeship

When it comes to delivering software, the accepted method of doing this is to make a build whenever something in your git repository changes. When the build is done and tested, you may release your software, for example - publishing it on GitHub as a versioned release, or by pushing a Docker image to the Docker Hub or your private registry. This process is called continuous integration.

For our example, we will take an ID generator service, which I will describe in more detail in the following chapter. I am going to use Codeship as the CI which will create the build and release the appropriate binaries to GitHub and push a release to Docker Hub.

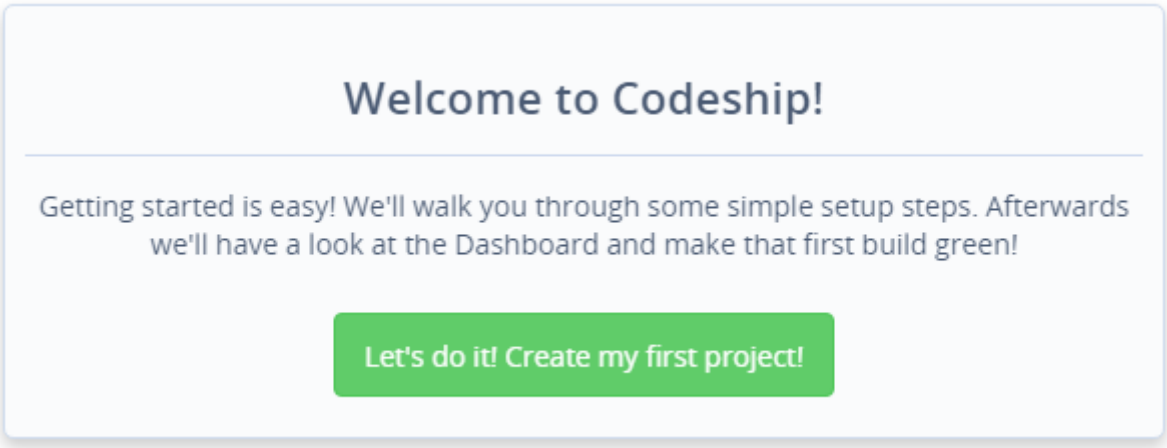
What will happen every time I push changes to GitHub is:

1. GitHub will notify Codeship that I made a push via Webhook,
2. Codeship makes a git checkout
3. Based on `codeship-services.yml` a build environment is created (software installed, etc),
4. Based on `codeship-steps.yml` - your software is built and deployed

There are some other parts within these steps slightly hidden away.

Starting with Codeship

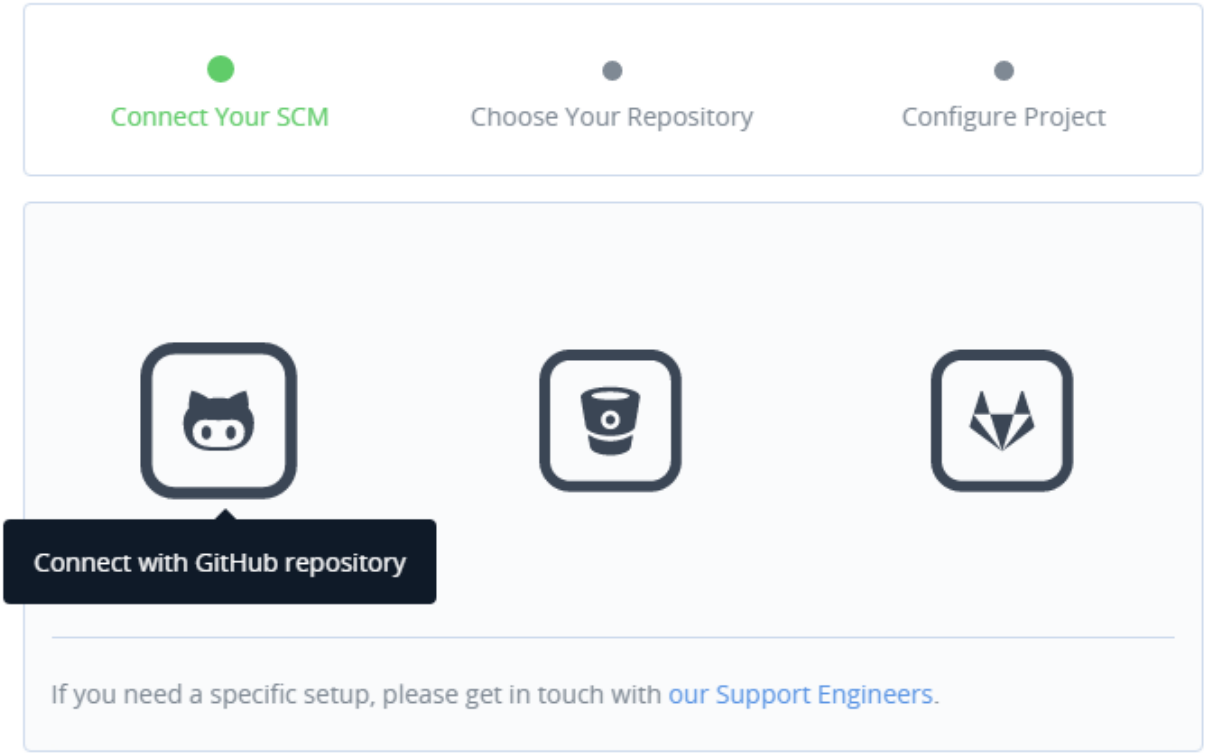
First off, signing up for Codeship[52] is simple, and their free tier gives you 100 builds per month. We will be using the "Codeship Pro" product, which has full Docker support, customizable CI environments and you can run a local build runner. Before getting ahead of myself, let me walk you through creating our first project:


Welcome to Codeship!
Getting started is easy! We'll walk you through some simple setup steps. Afterwards we'll have a look at the Dashboard and make that first build green!
Let's do it! Create my first project!

A welcome on your Dashboard

[52] https://codeship.com/

After clicking the big green button, Codeship will ask you which project you'd like to create. They support GitHub, Bitbucket and GitLab repositories out of the box. Our service, sonyflake[53] is hosted on GitHub, so I'm using that to connect to Codeship.


Connect Your SCM
Choose Your Repository
Configure Project
Connect with GitHub repository
If you need a specific setup, please get in touch with our Support Engineers.

A choice of SCM systems is available

When connecting your repository, be sure to choose the "Codeship Pro" plan. At this point, we will need to install their "Jet" CLI. There's full documentation available on their Running Codeship's Jet Locally for Development[54] page.

Why do we even need Jet?

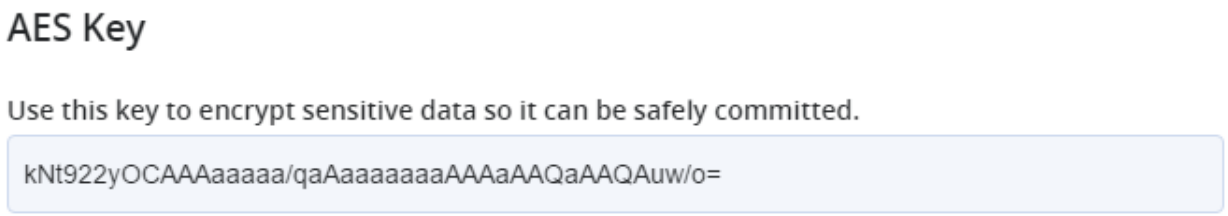
> Jet is used to locally debug and test builds for Codeship Pro, as well as to assist with several important tasks like encrypting secure credentials.

Very simply put, after you install Jet, you can run `jet steps` in your code checkout, to see how your build will perform on Codeship Pro. In development, you'll run it until you're satisfied that your build runs as it should - when you'll push your code, GitHub will trigger the build on Codeship.

Since I'm setting up a release to Docker Hub and to GitHub, I'll definitely need to securely add some credentials to the container where I'll be building our app. Codeship Jet provides a way to encrypt these credentials by providing an AES Key for the project.

[53] https://github.com/titpetric/sonyflake

[54] https://documentation.codeship.com/pro/getting-started/installation/


AES Key
Use this key to encrypt sensitive data so it can be safely committed.
kNt922yOCAAAaaaaa/qaAaaaaaaaAAAaAAQaAAQAuw/o=

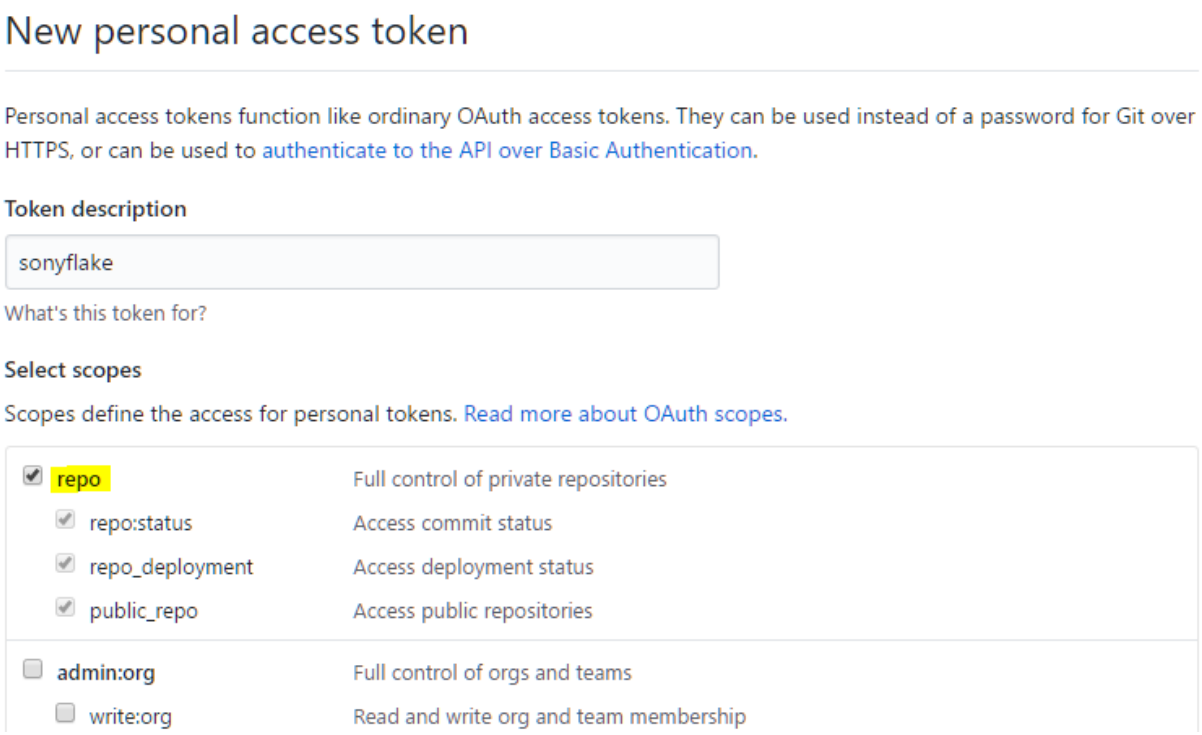
The AES key can be found under your project general settings

I saved the key as `codeship.aes` into the sonyflake repository, and made sure that I didn't commit it by mistake by adding it to the .gitignore file.

I will need a number of environment variables that will allow me to build and release sonyflake. Save these environment variables into a file named `.env`:

```
1 DOCKER_REGISTRY_USERNAME=titpetric
2 DOCKER_REGISTRY_PASSWORD=ImNotTellingYou
3 GITHUB_TOKEN=somethingsecret
```

The `GITHUB_TOKEN` should be generated on GitHub under "Developer Settings", "Personal Access Tokens", or by following this link directly[55]. The "repo" permission should be set for the token as it's marked on the screenshot.


New personal access token
Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.
Token description
sonyflake
What's this token for?
Select scopes
Scopes define the access for personal tokens. Read more about OAuth scopes.
repo
Full control of private repositories
repo:status
Access commit status
repo_deployment
Access deployment status
public_repo
Access public repositories
admin:org
Full control of orgs and teams
write:org
Read and write org and team membership

Creating a new token on GitHub

[55] https://github.com/settings/tokens/new

The other environment variables facilitate the login to the Docker Hub, so we can push our built docker image when we finish building it. Be sure to add `/.env` to the .gitignore file.

To encrypt these secrets, you can simply issue:

```
jet encrypt .env .env.encrypted
```

This will use your `codeship.aes` key to encrypt your environment variables, and after you do this, you can add and commit the `.env.encrypted` file. The file will be used from codeship-services.yml definition.

Services with Codeship

Your `codeship-services.yml` declares a service, which is used to run CI/CD builds with Codeship. Each of these services will produce a docker image, which will then be used to run individual steps in `codeship-steps.yml`. As our service is written in Go, we will use the builder pattern to produce a docker image that will contain only the statically compiled executable.

We will also publish the binaries to GitHub, so we will need to have github-release[56] as a dependency.

```
FROM golang:1.8-alpine

MAINTAINER Tit Petric <black@scene-si.org>

RUN apk --update add bash make docker && go get -u github.com/aktau/github-relea\
se

WORKDIR /go/src/app
```

For building the actual application, Dockerfile.build is used - in it, all the development-time dependencies like `make` or others are specified. Even docker is such a dependency, as we need to include it to enable the build a docker image.

As I declare the service for our build environment, I chose to use Codeship docker integration feature[57] to provide a docker environment to the build container. The service file for snowflake looks like this:

[56] https://github.com/aktau/github-release

[57] https://documentation.codeship.com/pro/getting-started/services/#docker-inside-docker

```
sonyflake:
  build:
    dockerfile: Dockerfile.build
  add_docker: true
  encrypted_env_file: .env.encrypted
  volumes:
    - ./:/go/src/app
```

Looking at both the `Dockerfile.build` above, and the declaration of the sonyflake service, you will notice that the source code for sonyflake is passed via the services `volumes` option. This enables us to run multiple steps later, which will use the same volume. So, we can separate our build step which will produce binaries from our deploy steps which will publish them.

CI Steps with Codeship

To use the service that we declared, we need to set up a `codeship-steps.yml` file. This file will declare all the commands that we want to run in our pipeline. I am creating three steps that will run in sequence. I am separating my "build" into two steps - as I'm interested in how much time compiling my application takes, and I'm also interested in how much time creating a docker image takes. I will see this information via the Codeship dashboard.

```
- type: serial
  tag: master
  service: sonyflake
  steps:
    - name: 'prepare'
      command: ./codeship-build.sh prepare
    - name: 'build go'
      command: ./codeship-build.sh build-go
    - name: 'build docker'
      command: ./codeship-build.sh build-docker
    - name: 'release'
      command: ./codeship-release.sh
```

The `tag: master` line is important here. We want to target only the commits/pushes that happen on this branch. When we push files to GitHub releases a new tag is created in the repository - this in turn triggers another webhook event that triggers another Codeship build. In order to prevent an infinite loop, this line was added.

I did wrap my logic into two scripts. The script `codeship-build.sh` basically invokes any possible step from the Makefile, while at the same time providing additional information that will be burned into our release container.

```
#!/bin/bash
set -e

## Get git commit ID
CI_COMMIT_ID=${CI_COMMIT_ID:-$(git rev-list HEAD --max-count=1)}
CI_COMMIT_ID_SHORT=${CI_COMMIT_ID:0:7}

## Get latest tag ID
CI_TAG_ID=$(git tag | tail -n 1)
if [ -z "${CI_TAG_ID}" ]; then
  CI_TAG_ID="v0.0.0";
fi
CI_TAG_AUTO="${CI_TAG_ID}"
if [ -f "build/.date" ]; then
  CI_TAG_AUTO="$(echo ${CI_TAG_ID} | awk -F'.' '{print $1 "." $2}').$(<build/.da\
te)"
fi

make -e CI_TAG_ID=${CI_TAG_ID} \
     -e CI_TAG_AUTO=${CI_TAG_AUTO} \
     -e CI_COMMIT_ID=${CI_COMMIT_ID} \
     -e CI_COMMIT_ID_SHORT=${CI_COMMIT_ID_SHORT} "$@"
```

The `Makefile` itself contains all the build commands that produce our binaries.

```
all:
  @echo 'Usage: make <prepare|build-go|build-docker>'

build-go: build/sonyflake build/sonyflake.exe
  @echo "Build finished"

build/sonyflake:
  CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o build/sonyflake main.go
  cd build && tar -zcvf sonyflake_linux_64bit.tgz sonyflake && cd ..

build/sonyflake.exe:
  CGO_ENABLED=0 GOOS=windows GOARCH=amd64 go build -o build/sonyflake.exe main.go
  cd build && tar -zcvf sonyflake_windows_64bit.tgz sonyflake.exe && cd ..

build-docker:
  docker build --rm -t titpetric/sonyflake --build-arg GITVERSION=${CI_COMMIT_ID\
} --build-arg GITTAG=${CI_TAG_AUTO} .
```

```
18
19 prepare:
20   @rm -rf build && mkdir build
21   @date +"%y%m%d-%H%M" > build/.date
22   @echo "Build folder prepared"
23
24 .PHONY: all build-docker prepare
```

With the declared targets, we do a little bit of Go magic - we produce a statically linked binary for 64bit Linux which we will also pack into a docker image. We also provide a 64bit windows executable, which we will push as a release to GitHub.

```
1  FROM alpine:3.5
2
3  MAINTAINER Tit Petric <black@scene-si.org>
4
5  ARG GITVERSION=development
6  ARG GITTAG=development
7  ENV GITVERSION=${GITVERSION} GITTAG=${GITTAG}
8
9  ADD ./build/sonyflake /sonyflake
10 ENTRYPOINT ["/sonyflake"]
```

As I was saying - it's a good idea to tag your docker images with commit hashes. The commit hash is an indicator which commit from git was used to build the image which you are pulling. This hash could be used to "lock" your deployment to a specific version, or it could be used to revert some breaking changes in your latest builds. Either way - useful.

The release script provides the same git variables but they are used to provide GitHub release tags and to tag our produced docker image and push them to GitHub. I'm going to break down the file `codeship-release.sh` to better illustrate the individual steps of the release process.

```
1  #!/bin/bash
2  set -e
3
4  ## Get git commit ID
5  CI_COMMIT_ID=${CI_COMMIT_ID:-$(git rev-list HEAD --max-count=1)}
6  CI_COMMIT_ID_SHORT=${CI_COMMIT_ID:0:7}
7
8  ## Get latest tag ID
9  CI_TAG_ID=$(git tag | tail -n 1)
10 if [ -z "${CI_TAG_ID}" ]; then
```

```
11    CI_TAG_ID="v0.0.0";
12  fi
13  CI_TAG_AUTO="$(echo ${CI_TAG_ID} | awk -F'.' '{print $1 "." $2}').$(<build/.date\
14  )"
```

The beginning of the release script takes care of generating the latest commit IDs and a semver-style tag starting at v0.0.0 if no tags already exist in your git repository. This information is used as somewhat of an audit trail that will be used to tag the docker image, as well as the release on GitHub.

```
1  ## Login to docker hub on release action
2  if [ ! -f "/root/.docker/config.json" ]; then
3    docker login -u $DOCKER_REGISTRY_USERNAME -p $DOCKER_REGISTRY_PASSWORD
4  fi
```

We are using the environment from our encrypted `.env.encrypted` file to log into Docker Hub. This enables us to use `docker push` from the release step.

```
1  function github_release {
2    TAG="$1"
3    NAME="$2"
4    latest_tag=$(git describe --tags `git rev-list --tags --max-count=1`)
5    comparison="$latest_tag..HEAD"
6    if [ -z "$latest_tag" ]; then
7      comparison="";
8    fi
9    changelog=$(git log $comparison --oneline --no-merges)
10   echo "Creating release $1: $2"
11   github-release release \
12     --user titpetric \
13     --repo sonyflake \
14     --tag "$1" \
15     --name "$2" \
16     --description "$changelog"
17 }
```

This is a bash function that uses `github-release` to create a new release on GitHub. For a bit of magic, a short "one-line" diff is added as the description of the release. This provides a minimal changelog, which gives a short overview of the changes between releases, listing all the commits and linking them to the commit detail view.

```
function github_upload {
  echo "Uploading $2 to $1"
  github-release upload \
    --user titpetric \
    --repo sonyflake \
    --tag "$1" \
    --name "$(basename $2)" \
    --file "$2"
}
```

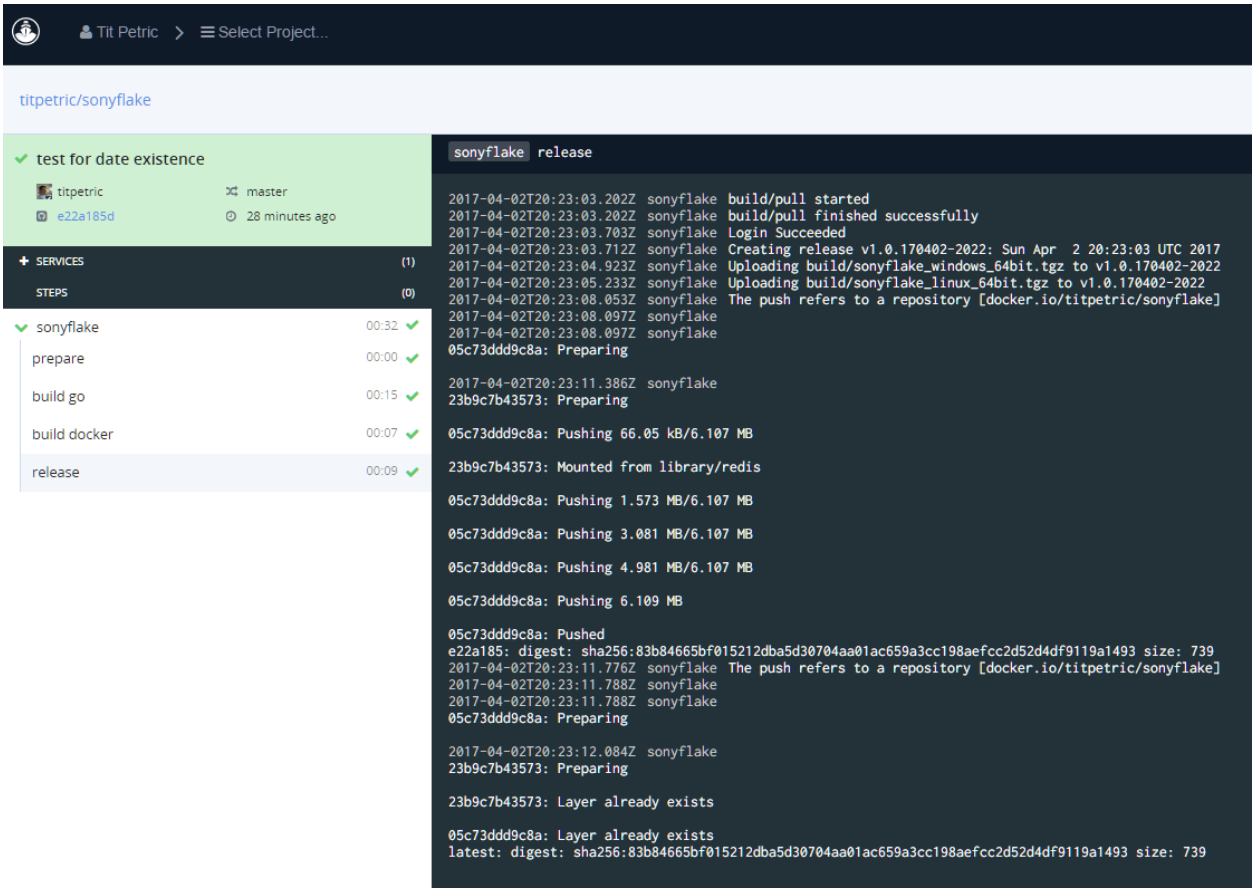
The function `github_upload` adds a new file to the release. This should be invoked once for every file that you want to upload.

```
## Release to GitHub
github_release ${CI_TAG_AUTO} "$(date)"
FILES=$(find build -type f | grep tgz$)
for FILE in $FILES; do
  github_upload ${CI_TAG_AUTO} "$FILE"
done
```

We create a new release, tagging it with `$CI_TAG_AUTO`. We add the files that we created in the build step to the release, with calls to `github_upload`.

```
## Release to Docker Hub
docker tag titpetric/sonyflake titpetric/sonyflake:${CI_COMMIT_ID_SHORT}
docker push titpetric/sonyflake:${CI_COMMIT_ID_SHORT}
docker push titpetric/sonyflake:latest
```

And the final part of the release - we tag our image with the short commit ID, and we push it and the latest image to Docker Hub. This way we can ensure that latest will always be up to date, while individual tags can be used for locking an image when you need to run a specific version.

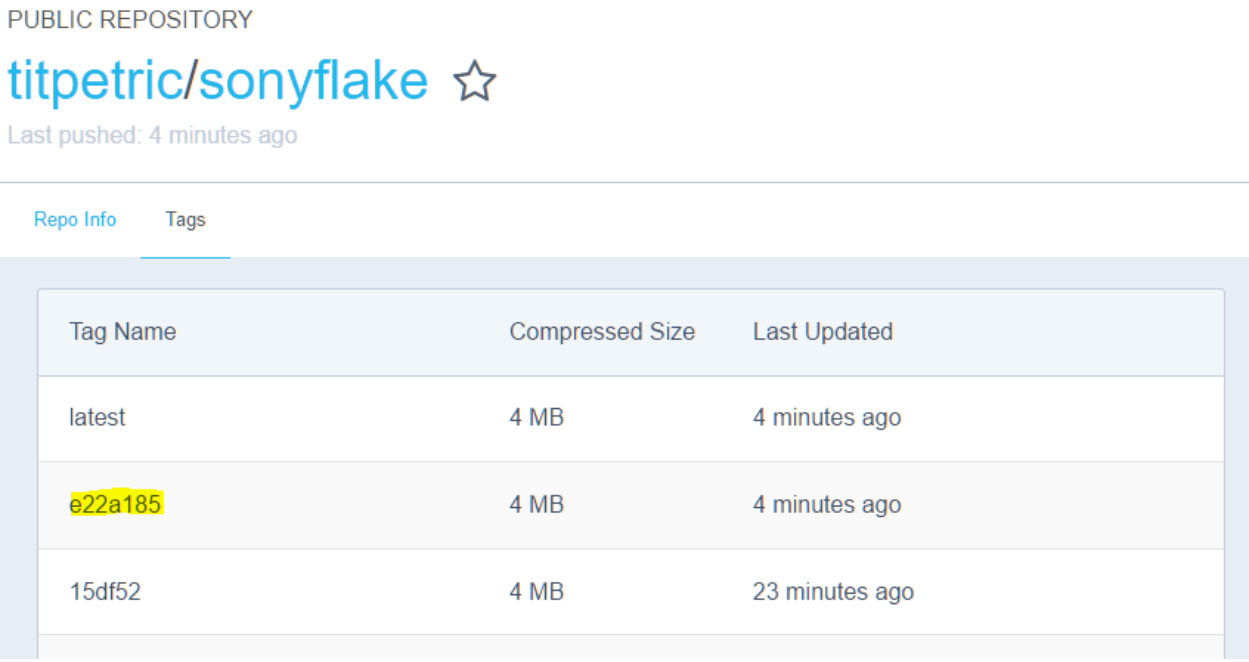


In a blink of an eye, I found my build passing

As we added various information from GIT to our container, we can inspect this information from the containers environment. We can override the entrypont to run something else in the container - `env` for example. This program will print the existing environment variables:

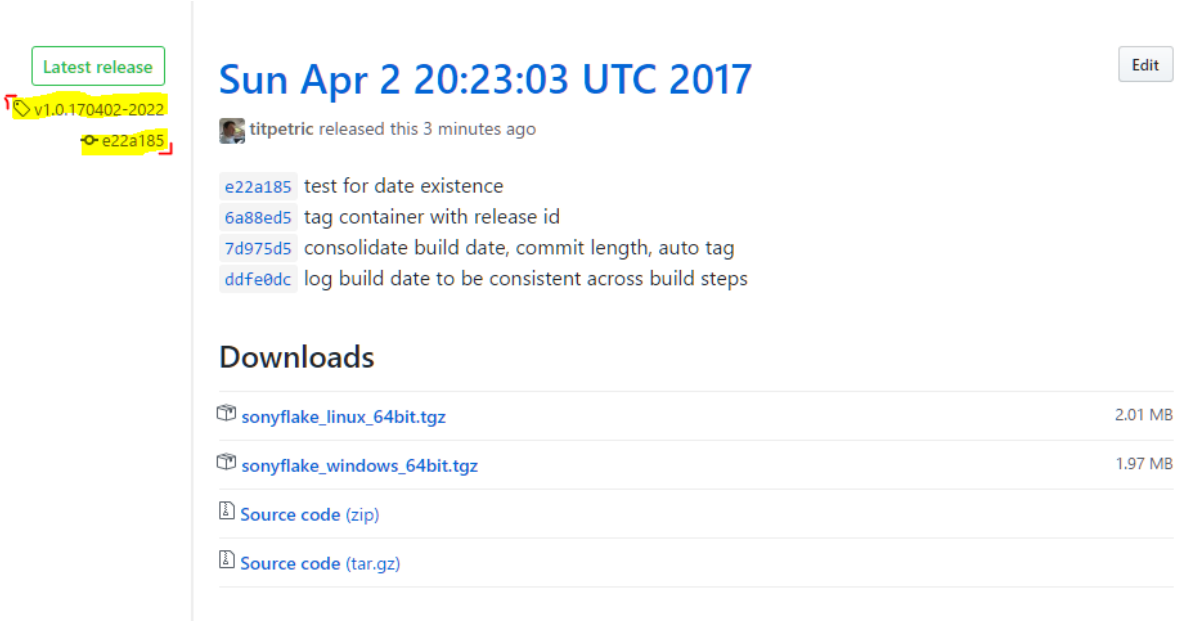
```
# docker run --rm --entrypoint=env titpetric/sonyflake | grep GIT
GITVERSION=e22a185d64838209ee3f62faf322fa0e8add3f70
GITTAG=v1.0.170402-2022
```

So, if you have a running container, you know exactly which version it's running. You can run the exactly same container on another host by using the GITVERSION above to pull/run the tagged image.



Finding a container on Docker Hub by commit ID

If you want just the binary, you can use the GITTAG value to find the release on GitHub and download that. This information enables you to be flexible with your releases. The GITVERSION matches both the docker registry and GitHub, and GITTAG matches a compiled GitHub release if you just need the binary.



Finding a release on Github by Tag/Commit ID

VI. Processes - Execute the app as one or more stateless processes

The principle says that the application itself shouldn't keep a local state - a local cache or in-memory values that should persist after shutdown of your application. An application which keeps state and has persistence is considered a backing-service.

When it comes to backing services, applications that store data have additional concerns that fall beyond the requirements of the 12 Factor App. These concerns may be how the data is persisted on disk (if at all), how the data is shared or replicated between instances, and what kind of consistency guarantees the service provides - like "eventually consistent" or in terms of databases, ACID (Atomicity, Consistency, Isolation, Durability) guarantees.

The 12 Factor App guidelines however envision an application which may be scaled to thousands of instances without much effort. In order for that to work, each application has to be able to start just with the configuration it receives from the environment. In contrast with backing services - a database can be scaled up if needed, but it usually involves taking snapshots, restoring them on new instances, setting up data replication and additional tasks that increase complexity and set up time. With a 12FA service, it's literally as simple as "run 1000 instances of this".

The additional intent of this guideline is to improve the reliability of your service in face of random outages. If individual applications don't have their own caches or data, it's more likely that they can tolerate unexpected crashes or outages without and impact.

All long-lasting or persistent state should come from your environment, or from backing services. The guideline doesn't forbid you to keep any local state, it however requires that any state you have should be restored exclusively from your environment or the backing services which take care of persistent data storage (RDBMS, NoSQL, Queues,...)

This chapter is closely related to Chapter IX. Disposability in the sense that there's just one logical conclusion towards testing it - reap some of the running application processes in order to discover possible problems in terms of caches and data storage.

Data caching

Data caching is a form of storage that would require your application to load/save a set of data which is frequently used or expensive to calculate. This has an implication about start up speed - your application takes time to load data, which is wasting resources that could already be used to serve requests. In the same way - if your application takes a lot of time calculating some values on startup, it is wasting valuable time - the data should just be retrieved from a backing-service as needed, and your app startup should be as fast as possible. This saves RAM and CPU resources.

Share-nothing

When we're talking about scalable and distributed systems, a phrase that often occurs is shared nothing architecture (SN)[58]. Specifically, this means that none of your applications may share memory or disk storage, nor use some kind of Inter-process communication[59] to talk and share data with each-other on the same host.

Think of it this way - your application container could go away without notice. As other processes (and starting up a new process) don't rely on any kind of shared data, a new one will be started with all the same properties. When data is shared in ways that don't scale (like the file system, for example), there's the possibility of cascading failure.

[58] https://en.wikipedia.org/wiki/Shared_nothing_architecture
[59] https://en.wikipedia.org/wiki/Inter-process_communication

Backing service: Running a highly available Redis instance

There are steps that you can take to move Redis closer to a 12 Factor App in terms of how it's deployed and managed. For example: you can turn off disk persistence and only store whatever you have in RAM. You can shard your data to better tolerate failures - if one instance goes down, you only loose 1/Nth of your data pool. If you only use it for caching (and not persistent storage), it's becomes a 12FA conforming app as instances may be killed off and started up by additional demand.

In terms of writing 12 Factor Apps, we need to consider how to create backing services which are also resilient to failure. If you rely on running your Redis instance locally on the same host that runs your application it gives additional guarantees of non-failure, mainly that a network outage doesn't affect connectivity between your app and a Redis instance.

When running redis in docker swarm mode, we can assume we have 3 docker swarm hosts. One of the hosts can go offline, taking a part of your redis data with it - but in terms of network connectivity, one out of three IPs is not reachable any more. Your app must be designed to handle such scenarios, and preferably eject one of the 3 IPs and keep using the remaining two until it comes back online.

The Docker hosts could have some sort of IP fail-over scenario, where you'd have a set of three floating IPs that would migrate between the hosts. As such, you can rely on connectivity on any of the Docker Swarm IPs to connect to your Redis service remotely. This is especially useful in cases where you want your backing service to be reachable even outside of your Docker environment, i.e., from dedicated virtual machines or external applications in 3rd party environments.

An interesting approach to using Redis as a backing service in Swarm is to connect only to the local instance of the running Docker daemon. This requires that an instance of Redis is running as a swarm service. Docker will route traffic from all swarm nodes on exposed ports to the actual nodes that are running the service. This way, you can be sure that connecting to your docker host IP will always give you connectivity to your backing services. It's quite elegant really.

> When you publish a service port, the swarm makes the service accessible at the target port on every node, regardless of whether there is a task for the service running on that node or not. This is less complex and is the right choice for many types of services.
>
> Publish ports - official docker documentation[60]

[60] https://docs.docker.com/engine/swarm/services/#/publish-ports

```
+----------------+
|Your application|
+----------------+
  +-----------+              +-------------------+
  |Docker host+ - - - - - -> +Another docker host|
  +-----------+              +-------------------+
                               +---------------+
                               |Redis container|
                               +---------------+
```

Your app could connect to the remote docker host (or hostname where you know Redis is running), but you'd have to configure things like IP failover, and you couldn't take a good advantage of docker swarm workers, as the swarm masters you would be connecting to will be forwarding traffic on to the correct host. Using your local Docker host is reliable, and is one "hop" less for your connection.

The obvious issue is, that you can't just connect to `127.0.0.1`. You need to get the IP of the docker host where your instance is running on. Since we use a custom network on `172.25.0.0/24` we know that our docker host will be listening on `172.25.0.1`. In case we didn't know this, we could run something like this to find out the IP:

```
# docker run -it --rm --net=party debian /sbin/ip route|awk '/default/ { print $\
3 }'
172.25.0.1
# docker run -it --rm debian /sbin/ip route|awk '/default/ { print $3 }'
172.17.0.1
```

As you can see, the command detects the IP of your Docker host even when you're not using a custom network (`--net=party` in our case). As long as you expose a Redis port in the Docker swarm, the port will be reachable on every host and will route your connections to a random running container.

> As your data outages might still hurt you, and perhaps you'd want to scale your Redis instances to many CPUs on the same host, you might simply either listen to multiple ports, or even put a dedicated proxy in front of your individual redis instances. Twemproxy[61] aka nutcracker can provide you the same interface as Redis (or Memcached) and proxy your connections while sharding all the data between multiple redis instances. This way, you can take advantage of your available RAM much more efficiently. This is usually a consideration of not only how one can make backing services highly available, but also resilient to failures.

[61] https://github.com/twitter/twemproxy

Realistic example of a stateless application

The best example of a true stateless application that I could come up with was known as twitter snowflake[62]. While I didn't dig around for the original code, Sony did make their own generator of 64bit IDs, sony/sonyflake[63]. It generates IDs similar to Twitter Snowflake, by combining the current timestamp, accurate to 10ms, a sequence number (up to 255) and a machine ID (up to 65535). This means it can generate about 25,600 IDs in a second from a single instance. As the IDs are uncoordinated, it means that any of the ID generator containers can be killed off, only reducing the overall capacity of the system, but not interrupting functionality.

I created a simple HTTP server built around sonyflake[64] that is suitable to be run in a docker or swarm environment. The original repository provides an example for running the service on AWS. The main difference is that this server uses the default machine ID which is created from the network interface the container is running on.

Sonyflake uses:

```
39 bits for time in units of 10 msec
 8 bits for a sequence number
16 bits for a machine id
```

So when you run a container, and it ends up on the IP "172.5.0.11", your machine ID will be "11", or differently - the lower 16 bits of the private IPv4 address. You can combine as many containers as you need, the limitation is just that the machine IDs will change on each instance. As we're attaching the service to a common network, the machine IDs will be as unique as the network the containers run on.

When it comes to running software in a docker service, the traffic is distributed between replicas. As a replica goes away, Docker Swarm makes sure that a new container starts up on one of the hosts available. This provides a level of reliability in case of outages. However - when containers unexpectedly go away, they may do so after a connection to them is established, but before the response is sent. This is why usually people run load balancing software in front of worker pools - in case a worker starts misbehaving and returns an invalid response (timeout, unexpected disconnect, non 200 http response,...) then the load balancer can retry the same request on a different, adjacent container.

In order to start sonyflake as a docker swarm service (exposed on port 80):

[62] https://blog.twitter.com/2010/announcing-snowflake
[63] https://github.com/sony/sonyflake
[64] https://github.com/titpetric/sonyflake

```
# docker service create \
  --replicas 5 \
  --network party-swarm \
  --update-parallelism 5 \
  --name sonyflake \
  -p 80:80 titpetric/sonyflake

pc70i7o8vl7kbn9p9vq14tqt1
# curl -s 127.0.0.1
{"id":134164003092955146,"machine-id":10,"msb":0,"sequence":0,"time":7996797746}
```

It really is that simple to start a distributed fault-tolerant service. We can benchmark the ID generator with another container, this time I'm going to use the public williamyeh/wrk[65] image.

```
# docker run --net=party-swarm --rm williamyeh/wrk -t 6 -c 30 http://sonyflake
Running 10s test @ http://sonyflake
  6 threads and 30 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   774.63us    2.01ms  68.74ms   96.38%
    Req/Sec     4.34k     1.78k    9.76k    68.33%
  259166 requests in 10.01s, 50.23MB read
  Socket errors: connect 0, read 0, write 0, timeout 56
Requests/sec:  25882.99
Transfer/sec:      5.02MB
```

So 10 seconds of generating IDs over 5 containers (and only 3 swarm hosts) generated about 260 thousand ids. The number of requests that managed to run here are still below the theoretical threshold of how many IDs even a single sonyflake instance is able to produce.

The sonyflake service as created above may be used as a backing service for your app. When it comes to scaling your data, you may partition it on the value of `machine-id`. Or you may just rely on the uniqueness of the IDs to provide a master-master replication of MySQL as a backing service.

The requests to the containers are generally distributed quite equally in the default VIP (Virtual-IP) mode:

[65] https://hub.docker.com/r/williamyeh/wrk/

```
# while true; do curl -s 127.0.0.1 | jq '.["machine-id"]'; done | head -n 1000 |\
 sort | uniq -c
    200 10
    200 3
    200 6
    199 7
    201 8
```

When you're dealing with keep-alive connections, they will stay open longer and the load may be unevenly distributed between your back-end containers. Smarter logic is needed to route such requests and docker doesn't provide much towards this end. The only option you have is how the addressing to the service is done with the `--endpoint-mode` flag.

> You can configure the service to use DNS round-robin directly without using a VIP, by setting the –endpoint-mode dnsrr when you create the service. DNS round-robin is useful in cases where you want to use your own load balancer.

> The principles of Docker Swarm are explained in detail in Chapter VIII. - Concurrency. I'm using the above example to demonstrate a service which keeps absolutely no local state and can be scaled with next to zero effort.

VII. Port binding - Export services via port binding

When it comes to Docker and networking, there are several modes of operation that are important for the containers you run. You may choose to run your containers on the same network as the docker host, you may isolate the network all together. This behaviour is determined by the network driver which you use.

With docker, you are enabled to define your own, user-defined networks. You can create these networks on single hosts or across a cluster of hosts (docker swarm). By default docker comes with several networks created:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

The none network

The `none` network doesn't add any network interfaces to your container. This means that whatever code that's running in your container can't use any network capabilities. The container can't connect to the internet, and no services from the container can be exposed to other containers.

```
# docker run --rm -it --network=none alpine:3.5 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
```

As you see, only the "lo" (loopback/localhost) interface is available in the container. This networking mode (or lack thereof) is usable when you're considering running untrusted code, where you don't want to have any access to the internet. Or just because you know that some CLI application you're running should not have access to the internet.

For example, video processing jobs with FFMpeg would consume what is available on a filesystem, and produce output on a filesystem. In such cases, there is process control around FFMpeg so we know if the encoding job finished successfully. By itself, FFMpeg does not require any access to the network. Such jobs require a process management that runs the FFMpeg containers as needed.

The bridge network

The bridge network is the default network when you're running docker containers on a single host. This is the network that is routed over the `docker0` interface on your docker host.

If you run multiple containers on the default network, they do not see each-other via DNS, but you can ping/connect to other containers, if you know their IP.

```
# docker run -itd --name sh1 alpine:3.5 sh
cfa22b4df2b5eb06dd0621c29a880b250cca35485ddc7d148eec39168e3f3913
# docker run -itd --name sh2 alpine:3.5 sh
f3e4944530fbfa0cc37ae53e388df53b532e7c4efbc56e27cdcab1d95840084b
# docker run --rm -it alpine:3.5 ping sh1
ping: bad address 'sh1'
```

But, you can ping/connect to other containers on the same network if you know their IP.

```
# docker exec -it sh1 ip addr | grep global
    inet 172.17.0.3/16 scope global eth0
# docker exec -it sh2 ip addr | grep global
    inet 172.17.0.4/16 scope global eth0
# docker exec -it sh2 ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.167 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.111 ms
^C
```

The internet is freely accessible from these containers, so you may for example, fetch packages from the internet, consume public API services or connect to other hosts in the same private-lan.

Custom bridge network

You can create a number of custom bridge networks for your containers. In the introduction of this book, we create our custom bridge network `party`, like this:

Create a custom bridge network

```
#!/bin/bash
#
# this creates a custom docker "bridge" network named "party"
#
docker network create -d bridge --subnet 172.25.0.0/24 party
```

There's a key difference between this network and the default `bridge` network. The containers that will run on the `party` network can immediately be visible to other containers on the network via internal docker DNS service. Recreating the example with two containers:

```
# docker run -itd --name sh1 --network party alpine:3.5 sh
80e2c534a6f41c7c8b1a4347dfadf733eab6564f05282e589235cb9fb853a47f
# docker run -itd --name sh2 --network party alpine:3.5 sh
f8580d25aee73153ef5247de0330205d8aae50e36232ef6466ad713ad84d913d
# docker exec -it sh2 ping sh1
PING sh1 (172.25.0.2): 56 data bytes
64 bytes from 172.25.0.2: seq=0 ttl=64 time=0.167 ms
64 bytes from 172.25.0.2: seq=1 ttl=64 time=0.110 ms
^C
```

This network is however visible only on a single docker host.

Multi-host network

One driver which wasn't mentioned so far is the `overlay` network driver.

> Using docker engine running in swarm mode, you can create an overlay network on a manager node.
>
> The swarm makes the overlay network available only to nodes in the swarm that require it for a service. When you create a service that uses an overlay network, the manager node automatically extends the overlay network to nodes that run service tasks.

This multi host network is suitable only in LAN environments. Please don't use it to pair docker swarms in different regions or cloud providers. Doing that will have consequences on the account of that the networking layer which Docker daemons use to communicate with each-other needs to provide consistent latency. When deploying a multi-region cluster, each running swarm should be independent from other regions.

In order for containers to be discoverable between hosts in a docker swarm, we will need to create an `overlay` network on one of the manager nodes.

```
# docker network create \
  --driver overlay \
  --subnet 10.0.0.0/20 \
  --attachable \
  party-swarm
```

Since we can use a higher number of containers, we can declare a larger number of IPs available using the CIDR notation[66]. If we used `/24` we would have $2^{(32-24)} = 2^8 = 256$ IPs available. Using `/20` gives us 2^12 (4096) possible IPs and by extension, containers, that can run on this network.

The switch `--attachable` creates this network in a way, which may be used from non-swarm containers. This provides access to the network from the classic `docker run` way of running your containers.

I will create two containers on different hosts, and resolve them from the current host on which I am living on.

[66] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

```
# ssh swarm2 docker run -itd --name sh1 --network party-swarm alpine:3.5 sh
6367bce89b0e255214456f4063c29b21bd8e573f86dc1c15ee0d5c6ce7114b99
# ssh swarm3 docker run -itd --name sh2 --network party-swarm alpine:3.5 sh
d84267c6b277216d0a4363fc2a731d56af5b480b028a6816d68941eda13c3d41
# docker run -it --rm --network party-swarm alpine:3.5 nslookup sh1
Name:      sh1
Address 1: 10.0.0.8 sh1.party-swarm
# docker run -it --rm --network party-swarm alpine:3.5 nslookup sh2
Name:      sh2
Address 1: 10.0.0.9 sh2.party-swarm
```

Actually, these containers are part of two networks, the `party-swarm` network and another system network: `docker_gwbridge`. This network was created together with docker swarm, and occupies the "172.18.0.0/16" subnet by default.

```
# ssh swarm2 docker exec sh1 ip addr | grep global
    inet 10.0.0.8/20 scope global eth0
    inet 172.18.0.5/16 scope global eth1
```

The intent of this extra network is to provide connectivity outside of the cluster, and is created on each docker swarm node.

Host networking

When you run a service on the host network, the container has full access to the host networking stack. This means that the container may control creation, modification and even deletion of network interfaces. This is very useful in the case of VPN clients, network sniffing like tcpdump, or providing higher level networking tools and monitoring of your network infrastructure.

The caveat is also that any service that listens to a port inside the container suddenly becomes public. Basically, the containers network isolation is turned off, meaning if you would run redis with `--net=host` mode, the port it exposes is opened to anyone which has access to the IP of your host. System administration becomes a bigger factor in this case, much like it would if you would install software in the host directly (while still keeping the benefits of filesystem isolation, etc.).

> Basically, don't use host networking, unless you're *absolutely* sure what you're doing.

Exposing your service ports

The suggested way to run your service is to attach your containers to a custom bridge or overlay network - depending if you invoke your containers from docker or as a swarm service. Both ways to run your software enable you to expose individual ports over their `-p` options if needed.

If you will look at the Redis example in Chapter IV., you will notice that no ports have been exposed. Redis is a private service, which runs on our `party` network. This means that other containers on the same network can access it, without docker having to set up some kind of proxy.

When you explicitly want to publish a service port it's only one option:

```
-p=[] : Publish a containerá%¿s port or
        a range of ports to the host

    format:
        ip:hostPort:containerPort |
        ip::containerPort |
        hostPort:containerPort | containerPort

    Both hostPort and containerPort can be specified as a
    range of ports. When specifying ranges for both, the
    number of container ports in the range must match the
    number of host ports in the range, for example:

        -p 1234-1236:1234-1236/tcp

    When specifying a range for hostPort only, the
    containerPort must not be a range.  In this case the
    container port is published somewhere within the
    specified hostPort range. (e.g., `-p 1234-1236:1234/tcp`)
```

So, if the application you are running listens on port 3000, but you want to expose port 80, you can pass `-p 80:3000`. This makes it easier to run applications which aren't completely 12FA compliant and you can't set their port via environment variables.

When it comes to exposing your application on a port where it should listen to, to completely follow 12FA guidelines - you must read the port on which you listen on from the environment, but you can provide a reasonable default. With docker you have the option to be lazy a bit, and expose the service on different ports as mentioned above.

A reasonable default for your service port should be the typical port on which your service would listen to in production. If you're writing a web server, your default port should be port 80. If you're

writing a SOCKS HTTP proxy, your port might be 8080. Depending on protocol and intent, your default port should reflect your service.

There are other considerations as well. When running software on Linux, ports below the number 1024 are privileged ports. This means that as a non-privileged user (i.e.. non-root), you can run a server only above these ports. This is part of the reason why Ruby runs on the default port 3000. If your application will run with reduced privileges, you'll have to figure out which port works for you best. In terms of production deployments, you will still need some sort of reverse-proxy that listens on standard HTTP ports (80, 443) and then forwards traffic to your back-end service.

Debugging docker network connectivity

When it comes to debugging network connectivity there are usually a couple of things that I check whenever I am having some issues. In Debian there's the package `net-tools` which provides me with the utility `netstat`. With it, it's exceptionally easy to check if a given service is listening to a port. For example, if you want to know all the internal service ports that your docker application is binding to, you can issue this:

```
# docker exec -it apt-cacher netstat -pan
Active Internet connections (servers and established)
Proto  Local Address     Foreign Address  State   PID/Program name
tcp    127.0.0.11:37391  0.0.0.0:*        LISTEN  -
tcp    0.0.0.0:3142      0.0.0.0:*        LISTEN  7/apt-cacher-ng
tcp6   :::3142           :::*             LISTEN  7/apt-cacher-ng
udp    127.0.0.11:32837  0.0.0.0:*                -
```

Optionally you can append `| grep LISTEN` to view only the ports which are listening for incoming connections. Here we have an instance of `apt-cacher-ng` listening on IPv4 and IPv6 on port 3142.

You may wonder what is the `127.0.0.11` IP/open port. It's dockers internal DNS service.

```
# docker exec -it apt-cacher cat /etc/resolv.conf
nameserver 127.0.0.11
options ndots:0
```

There are many tools available in Linux that aid with debugging network connectivity as well as program execution. It's good that you have at least basic understanding of the following tools:

- `telnet`, `netcat` - open a raw connection to an IP/host and port (TCP),
- `curl`, `wget` - issue HTTP requests against an endpoint,
- `netcat`, `ss` - list open connections, list listening sockets,
- `lsof` - list open files by programs running
- `strace` - print system trace from a program which you run

Especially strace might be useful when it comes to debugging network connectivity. I had a case where it would take several seconds before I would be able to connect to a database instance running in Docker. I did not know where this lag was coming from, so I tried everything.

In that case, `telnet` told me that it took several seconds before the connection to MySQL was open. This confirmed that it wasn't the issue with the database. Running a few other services like Redis and Nginx, I verified that all of them were subject to this lag.

Only with `strace` did I manage to make progress debugging this. The utility gives you a system call trace for any application which you run. I installed it and ran the same telnet line, to connect to my MySQL server again. The application gives a lot of verbose output with all the system calls that are performed. It's explanation is a bit out scope, but what we're looking for is any `socket` and `connect` lines.

```
# telnet db1 3306 | grep socket
```

The offending line that jumped out at the time of the delay was:

```
socket(PF_INET, SOCK_DGRAM|SOCK_NONBLOCK, IPPROTO_IP) = 3
```

This only tells us that the socket was created in SOCK_DGRAM mode - UDP. DNS functions on UDP. After looking at more output, specifically some `connect` lines, I managed to isolate the DNS used as 127.0.0.11 - the internal Docker DNS server. I spotted something suspicious in the output:

```
read(3, "search mtk.lan\nnameserver 127.0."..., 4096) = 53
```

I see that, somehow, the name resolution from one of my VPNs leaked into the Docker environment. The cause of the delay was being caused by software trying to resolve non-existent domains based on an outdated `/etc/resolv.conf`. I checked the config file on the host - as I wasn't connected to the VPN any more, the "search mtk.lan" option wasn't in the resolv.conf any more. But it was still available inside the docker container.

I removed the container, and restarted it from a clean image. The container still had this `search` option inside resolv.conf. The option was burned into Docker somewhere, and I don't know a way to get rid of it! After some searching, I've located a filed issue against Docker, which describes similar behaviour (Issue #8425[67]). The issue is closed but it gave me enough info to fix it for myself:

```
$ service docker restart
```

The above command shuts down all containers and restarts the Docker daemon. When the daemon started, it picked up the current resolv.conf from the host, and when it started the containers - the network connectivity issues have gone away.

Since then I've also pinpointed the same issue on my development environment. I removed all the `search` lines from resolv.conf and restarted Docker as mentioned above. All connectivity issues/delays have practically gone away. This is what my Docker resolv.conf looked like before, and after:

[67] https://github.com/docker/docker/issues/8425

```
$ docker exec -i dev cat /etc/resolv.conf
search mtk.lan
nameserver 127.0.0.11
options ndots:0
$ docker exec -i dev cat /etc/resolv.conf
nameserver 127.0.0.11
options ndots:0
$
```

It's good to know the basic networking and system tools so you can inspect software operation. Of course, many times prevention is better - I never set up hosts without monitoring of some kind. There are great software packages that I use and can recommend to set up monitoring and alerting:

- Grafana[68],
- Netdata[69],
- Prometheus[70]

[68] https://grafana.com/
[69] https://github.com/titpetric/netdata
[70] https://prometheus.io/

Using your own DNS

If you need to force use of your own internal DNS for whatever reason, docker provides an option to pass the DNS servers you want to the container.

```
# docker run -it --rm=true --dns 8.8.8.8 alpine:3.5 ping www.google.com
PING www.google.com (216.58.214.196): 56 data bytes
64 bytes from 216.58.214.196: seq=0 ttl=56 time=13.121 ms
64 bytes from 216.58.214.196: seq=1 ttl=56 time=14.559 ms
64 bytes from 216.58.214.196: seq=2 ttl=56 time=12.117 ms
64 bytes from 216.58.214.196: seq=3 ttl=56 time=13.643 ms
^C
--- www.google.com ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 12.117/13.360/14.559 ms
```

I'm using the excellent janeczku/go-dnsmasq[71] to run a development DNS server, where I override some public DNS entries to make my life easier for development. For example - a website that I develop has Facebook OAuth login integration. In order to test that, I need to override the DNS resolver so point the public domain to the development server.

```
#!/bin/bash
NAME="dnsmasq"
DOCKERFILE="janeczku/go-dnsmasq:latest"
PARENT=$(dirname $(dirname $(realpath -s $0)))
docker rm -f $NAME
docker run --restart=always -d -h $NAME --name $NAME -p 53:53/udp -p 53:53 -v $P\
ARENT/conf:/conf $DOCKERFILE -f /conf/hosts --verbose -n 8.8.8.8,8.8.4.4
```

The service has another great feature - the management of the entries for it is done with a hosts file. You can literally pass your `/etc/hosts` to go-dnsmasq and have a DNS server that resolves your hosts entries available anywhere. There is also a sweet candy available in dnsmasq, which isn't available in your hosts files: you can create wildcard entries:

```
10.1.1.2 *.scene-si.org scene-si.org
```

I use similar entries to the above to provide various entrypoints for my projects. These may be `api.scene-si.org`, `admin.scene-si.org` or whatever else I need. The best part is, that I can start using a subdomain right away, without having to add it to the hosts file or restarting the dnsmasq

[71] https://github.com/janeczku/go-dnsmasq

service. I just run dnsmasq once, and then set up a nginx reverse proxy that can handle any subdomain.

In fact, I even auto-generate most of the DNS entries based on the file structure on disk. I set up my web projects under `/var/www/[name]/[dns]`. Of course there's some glue that puts everything together - basically it's similar to jwilder/nginx-proxy[72] in that it detects which projects you have and sets up the configuration for the DNS entries.

[72] https://github.com/jwilder/nginx-proxy

Sending e-mails from docker

There's a few ways to quickly set up email - the most basic one is that you don't. For example, if you're using Debian, postfix, sendmail or whatever it has bundled already comes configured so that it tries to connect to a smtp host named `mail`. All you need to do is start up a mail container on the same network, or define it in a docker container with `--add-host`.

However, we can extend this behaviour to enable some additional functionality for our docker containers - enabling them to talk to exposed services on the docker host. These services might be in different networks and might just expose a public API via a network port.

```
--add-host docker:172.25.0.1 --add-host mail:172.25.0.1 --add-host $(uname -n):1\
72.25.0.1
```

With our custom `party` network, we can use the gateway `172.25.0.1` as the docker host, which it effectively is. Every docker container which has exposed ports via the `-p` option is reachable now from the container. If we know the host name we're running on, we can use it's qualified name (`uname -n`), or we can always use `docker` as the alias that points to our docker host.

If running your own mail server on the host is not an option, it's very simple to install `ssmtp` into any container which you're running. This is partly why I advocate basing your docker images on alpine and not on scratch - usually software needs other software.

```
# apk --update add ssmtp
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/community/x86_64/APKINDEX.tar.gz
(1/1) Installing ssmtp (2.64-r9)
Executing busybox-1.25.1-r0.trigger
OK: 4 MiB in 12 packages
```

Installing it is simple, it's relatively small and the most important part, you can set it up with only a few environment variables:

```
cat << EOF > /etc/ssmtp/ssmtp.conf
mailhub=$SSMTP_SERVER:$SSMTP_PORT
AuthUser=$SSMTP_USER
AuthPass=$SSMTP_PASS
UseSTARTTLS=$SSMTP_TLS
hostname=$SSMTP_HOSTNAME
FromLineOverride=YES
EOF
```

A fully featured example is available on this GitHub Gist - setup_ssmtp.bash[73]. The basic idea is, you should run it from your ENTRYPOINT before you start up your application. This way, you can move your container anywhere, or you can move your smtp server anywhere and all it needs is a container restart. This is Chapter III. at work.

Of course, for actual mail delivery you could use any number of public or private services. Depending on the scope of your service, you may use Google Mail SMTP services to send your emails, or look at services like Sendgrid[74] or Mailchimp[75]. If you feel like configuring your own alternative, Amazon SES[76] is a cost-effective option, and if you want to take out some ground work from using it, you should take a look at Sendy[77].

[73] https://gist.github.com/titpetric/114eb27f6e453e3e8849d65ca1a3d360
[74] https://sendgrid.com/
[75] https://mailchimp.com
[76] https://aws.amazon.com/ses/
[77] https://sendy.co

VIII. Concurrency - Scale out via the process model

While not applicable to Go programs directly, due to it's strong concurrency model and ability to scale it's processing to multiple CPU cores, it is worth to look at this option as a way to provide redundancy via the process model, and with this also a graceful upgrade path when deploying new application versions. By using Docker we can also satisfy basic process management requirements set forth in the 12FA guidelines - respond to crashed processes and handle user initiated restarts and shutdowns.

Basic process management

When it comes to making sure that your application is alive, docker has the built-in switch `--restart` available, which controls the life of your process after it may unexpectedly exit or if your VPS or server unexpectedly restarts.

When you're running your application, you have a few restart policies available:

- `no` - don't restart the container when the application exists (the default),
- `on-failure` - restart the container only if the application exists with a non-zero exit status,
- `on-failure[:max-retries]` - optionally set how many times the application may restart,
- `unless-stopped` - if you manually stop the container, it will not restart when docker starts up,
- `always` - restart your application when it exits or when docker starts up (like after reboot).

Obviously, the default for any application could be `always`, unless you have a really good reason for keeping the application unavailable. I'm an advocate of `docker stop` && `docker rm` which will erase the container completely, so even with an aggressive restart policy it will have no side-effects - like for example, spinning up an unprotected management container after a reboot of the docker host.

In special cases, the restart policy protects you from unexpected crashes of your instances. Usually, people start with running services like redis on the host via the `init.d` (system V init tools). That ensures that redis would start after the reboot of the host, but it would not ensure that redis would automatically restart in case of an unexpected crash. I've had the misfortune to be in just that situation where Redis would crash with an OOM (out of memory) error, while there is still a generous

amount of RAM available. GitHub issue on it was filed, but eventually I moved Redis to docker, where I can take advantage of this restart policy immediately.

We can even test our restart policy with Redis quite effectively. Redis provides a command `DEBUG SEGFAULT` which will simulate a crash, and we can see how docker will respond to it. Our Redis service (see Chapter V.) is already started with `--restart=always`, so we only need to issue the command to make it crash.

In one terminal, you can view docker events with `docker events`. In another terminal, you can run the provided `redis-cli` helper from `services/redis` like this:

```
$ ./redis-cli debug segfault
Error: Server closed the connection
```

The output in the console with docker events will be something like this:

```
root@verdana:~# docker events
2017-02-06T20:11:30+01:00 container exec_create: redis-cli debug segfault 557...\
81a (image=redis:3.2-alpine, name=redis)
2017-02-06T20:11:30+01:00 container exec_start: redis-cli debug segfault 557...8\
1a (image=redis:3.2-alpine, name=redis)
2017-02-06T20:11:30+01:00 container die 557...81a (exitCode=139, image=redis:3.2\
-alpine, name=redis)
2017-02-06T20:11:30+01:00 container start 557...81a (image=redis:3.2-alpine, nam\
e=redis)
```

I shortened the output for brevity. As you can see - docker executes redis-cli inside the redis container, and immediately after it logs a `die` event - after which it takes less than a second to restart the container. When we were experiencing crashes once every few days this would have been a good enough bandaid.

Consider this - if the crash of the service was about 2 seconds, and it would happen every 3 days, the availability of the system would be 99.993%. Considering we had 3 instances which wouldn't crash at the same time, the availability would be a slightly higher 99.998%. For posterity, the reported OOM crash issue is available on GitHub[78].

> Obviously, having a basic process manager is not a big feature of docker. There are projects like upstart[a] or Supervisord which I'm also covering below.
>
> [a] http://upstart.ubuntu.com/

[78] https://github.com/antirez/redis/issues/3045

Advanced process management

Supervisord

There is also external software available that does similar things what docker `--restart` policy provides here. A popular software to manage processes is called supervisord[79]. In relationship to docker, I've mostly seen it used to combine several applications inside one docker image, most notably - the Nginx web server and PHP scripting language (php-fpm). This goes against the of single-responsibility principle, so it is going against good practices (one app per container).

Supervisord allows you to restart individual services via a web interface, and there are even attempts to provide a clustering layer above it, to see and manage processes in your deployment.

When defining applications to run with Supervisord, you would create a configuration block which would be something similar to this:

```
[program:nginx]
command=/usr/sbin/nginx -g 'daemon off;' -c /etc/nginx/nginx.conf
directory = /etc/nginx
autostart=true
autorestart=true
priority=10
stdout_events_enabled=true
stderr_events_enabled=true
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

The various options like `autostart` and `autorestart` are equivalent to `--restart=always` with docker. The directory is analogous to `-w` docker switch and there are many other options available and documented in the supervisord documentation[80].

> In my opinion combining Nginx and PHP is fine, mainly because I haven't seen a Nginx/php deployment that would use dedicated servers for each service. There's some performance benefits here as well as Nginx can use unix sockets to talk to PHP. In Go (or NodeJS) it's very easy to create a HTTP server, so there is less need to tightly couple a web server in front of it as the application already performs both functions. The rule of thumb is to individually pack up services into docker

[79] http://supervisord.org/

[80] http://supervisord.org/configuration.html#program-x-section-values

> images - as such, at some point you could migrate from `apache` to `lighttpd`, and later to `nginx` - all while keeping the `php` image as it is.

PM2

Another process manager of note is PM2[81]. When it comes to this process manager, it serves a different purpose than supervisord. Since Node is a single-threaded application it means that it may in the best case utilize a maximum of one CPU core. Scaling beyond 1 CPU core should be done with a process manager.

PM2 is slightly more smart about process management, and provides some functionality of `docker swarm`, the obvious is that it can scale your service over many CPUs, the less obvious one is that it can provide zero-downtime restarts (rolling update).

Actually, there's no reason on gods green earth why PM2 couldn't be used on a single host to provide several instances of your Go application. Of course, when developing with Go, the goroutines already enable you to scale your application beyond 1 CPU, so the obvious benefit is mainly to provide a zero-downtime upgrade.

There's a small, 24 MB docker hub package available which packages PM2 and the Node runtime. We will use keymetrics/pm2-docker-alpine[82] to run our application example from the previous chapter (run `build.sh` from chapter7/ samples).

> Go applications are not exactly 'static'. With that in mind, I'm using `golang:1.8-alpine` to build the application, as the PM2 docker image is also built on alpine. This ensures that the required libraries will be available when running our Go application in a container. Ideally the application should be tested when building a container with it copied inside. We'll return back to this later in a following section.

To run our Go application with PM2, we will need to create a `process.yml` file. This file holds the configuration of our Go application and how it should run.

[81] https://github.com/Unitech/pm2

[82] https://hub.docker.com/r/keymetrics/pm2-docker-alpine/

A PM2 configuration file for running our Go app

```
apps:
  - script: ./gotwitter
    name: 'gotwitter'
    interpreter: none
    execute-command: true
    instances: 2
    exec_mode: fork
    watch: false
    out_file: /dev/stdout
    error_file: /dev/stderr
```

There are only a few notable options here:

- `exec_mode` is 'fork', meaning PM2 will run processes with exec (syscall),
- `instances` specifies how many processes to run

When it comes to Node apps, `exec_mode` can be set to `cluster`, which enables scaling only for Node applications, using the clustering features it provides. Unfortunately PM2 doesn't provide process scaling in fork mode, so you can't tell it to run extra processes without a full restart of PM2.

Running our Go app with PM2

```
apps:
  - script: ./gotwitter
    name: 'gotwitter'
    interpreter: none
    execute-command: true
    instances: 2
    exec_mode: fork
    watch: false
    out_file: /dev/stdout
    error_file: /dev/stderr
```

When we start our Go application with PM2, there are some options available over the PM2 command line. I added a wrapper for running PM2 in docker in `pm2_exec`. For example, we can list the status of our application with `./pm2_exec status`.

```
# ./pm2_exec status
┌─────────────┬────┬──────┬─────┬────────┬─────────┬────────┬─────┬──────────┬─────\
──────┐
| App name  | id | mode | pid | status | restart | uptime | cpu | mem      | wat\
ching |
├─────────────┼────┼──────┼─────┼────────┼─────────┼────────┼─────┼──────────┼─────\
──────┤
| gotwitter | 0  | fork | 20  | online | 0       | 91s    | 0%  | 4.0 MB   | ena\
bled  |
| gotwitter | 1  | fork | 24  | online | 0       | 91s    | 0%  | 4.1 MB   | ena\
bled  |
└─────────────┴────┴──────┴─────┴────────┴─────────┴────────┴─────┴──────────┴─────\
──────┘
```

Each application instance reads the NODE_APP_INSTANCE environment variable (id column), and starts the application on the default port + this value. App ID 0 runs on port 3000, ID 1 runs on 3001, and so on. Because we're running in fork mode, there is no load balancer provided for the application and scaling features are unsupported for such apps.

```
# ./pm2_exec scale gotwitter 4
[PM2][ERROR] Application gotwitter is not in cluster mode
```

What we can do is run a HAProxy instance which connects to individual proxy ports. You could even use Nginx, if you want a smarter reverse proxy, but for our needs a quick HAProxy example should be enough.

Quick and Dirty HAProxy configuration

```
1  global
2          maxconn 10000
3
4  frontend public
5          bind *:80
6          mode http
7          timeout client 30s
8          default_backend gotwitter
9
10 backend gotwitter
11         balance roundrobin
12         mode http
13         timeout server 5s
14         timeout connect 5s
15         server app1 pm2-gotwitter:3000 check
16         server app2 pm2-gotwitter:3001 check
```

And the startup script

```
#!/bin/bash
docker rm -f haproxy
docker run -d --net=party -p 81:80 --name haproxy -v $(pwd)/haproxy:/usr/local/e\
tc/haproxy haproxy:1.7-alpine
```

With HAProxy in place, and reachable on port 81, we can now safely restart the PM2 service with any updates to the Go application.

```
# ./pm2_exec restart all
Use --update-env to update environment variables
[PM2] Applying action restartProcessId on app [all](ids: 0,1)
[PM2] [gotwitter](0) ✓
[PM2] [gotwitter](1) ✓
┌─────────────┬────┬──────┬─────┬────────┬─────────┬────────┬─────┬──────────┬──\
────┐
| App name  | id | mode | pid | status | restart | uptime | cpu | mem      | wat\
ching |
├─────────────┼────┼──────┼─────┼────────┼─────────┼────────┼─────┼──────────┼──\
────┤
| gotwitter | 0  | fork | 64  | online | 1       | 0s     | 0%  | 4.1 MB   | dis\
abled |
| gotwitter | 1  | fork | 68  | online | 1       | 0s     | 0%  | 3.9 MB   | dis\
abled |
└─────────────┴────┴──────┴─────┴────────┴─────────┴────────┴─────┴──────────┴──\
────┘
```

If you're dealing with a Node app, PM2 might be a good fit for starting out, but from what I see, I honestly wouldn't recommend it beyond using it on a single node. Currently, there are several hundred issues opened against the PM2 project on GitHub, and when it comes to replacing PM2 with the above functionality, it would pretty much be done with a few lines of bash code.

> Evaluating software like PM2 is difficult. The strength of PM2 lies in developing Node applications and running Go applications from it seems to be little more than an afterthought. I suppose the problem with any other programming language which they support will be similar to Go, as only Node applications can take advantage of the provided cluster mode.

Docker swarm

Scaling your service has usually been in the domain of system operators, which installed servers and developers tweaking software when the load got high enough to warrant scaling. Soon enough you'd be looking at tens or even hundreds of instances which took a lot of time to manage. With the release of Docker 1.12, you now have orchestration built in - you can scale to as many instances as your hosts can allow. And setting up a docker swarm is easy enough.

Initialize swarm

First off - I'm starting with a clean docker `1.12.0` installation. I'll be creating a swarm with a few simple steps:

```
root@swarm1:~$ docker swarm init
Swarm initialized: current node (4i0lko1qdwqp4x1aqwn6o7obh) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-0445f42yyhu8z4k3lgxsbnusnd8ws83urf56t02rv1vdh1zqlj-9ycry5kc\
20rnw5cbxhyduzg1f \
    10.55.0.248:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-0445f42yyhu8z4k3lgxsbnusnd8ws83urf56t02rv1vdh1zqlj-09swsjxd\
z80bfxbc1aed6mack \
    10.55.0.248:2377
```

I now have a swarm consisting of exactly 1 manager node. You can attach additional swarm workers, or add new managers for high-availability. If you're running a swarm cluster with only one manager and several workers, you're risking an interruption of service if the manager node fails.

> In Docker Swarm, the Swarm manager is responsible for the entire cluster and manages the resources of multiple Docker hosts at scale. If the Swarm manager dies, you must create a new one and deal with an interruption of service.

As we're interested in setting up a two-node swarm cluster, it makes sense to make both nodes in the swarm be managers. If one goes down, the other would take it's place.

```
root@swarm2:~# docker swarm join \
> --token SWMTKN-1-0445f42yyhu8z4k3lgxsbnusnd8ws83urf56t02rv1vdh1zqlj-09swsjxdz8\
0bfxbc1aed6mack \
> 10.55.0.248:2377
This node joined a swarm as a manager.
```

To list the nodes in the swarm, run `docker node ls`.

```
root@swarm2:~# docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
4i0lko1qdwqp4x1aqwn6o7obh     swarm1    Ready   Active        Leader
9gyk5t22ngndbwtjof80hpg54 *   swarm2    Ready   Active        Reachable
```

Creating a a service

As you see, when adding a new manager node, it's automatically added but not promoted to the leader. Let's start some service that will perform something that can be scaled over both hosts. I will ping `google.com`, for example. I want to have `5` instances of this service available from the start by using the `--replicas` flag.

```
root@swarm2:~# docker service create --replicas 5 --name helloworld alpine ping \
google.com
31zloagja1dlkt4kaicvgeahn
```

As the service started without problems, we just get the id of the service which was started. By using `docker service ls` we can get more information about the running service.

```
root@swarm2:~# docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
31zloagja1dl  helloworld  5/5       alpine  ping google.com
```

Of course, as we're talking orchestration, the services in the examples are split between `swarm1` and `swarm2` nodes. You can still use `docker ps -a` on individual nodes to inspect single containers, but there's the handy `docker service ps [name]`.

```
root@swarm1:~# docker service ps helloworld
ID           NAME          IMAGE   NODE    DESIRED STATE  CURRENT STATE
5fxtllouvmd  helloworld.1  alpine  swarm1  Running        Running 7 min
cqvgixx3djh  helloworld.2  alpine  swarm2  Running        Running 7 min
99425nw3r4r  helloworld.3  alpine  swarm2  Running        Running 7 min
1dj3cs7v5ij  helloworld.4  alpine  swarm1  Running        Running 7 min
0hy3yzwqzln  helloworld.5  alpine  swarm1  Running        Running 7 min
```

Testing fault tolerance

As we connected two managers to run our service, let's just bring one of them down. I'm going to power off `swarm1`, the current leader, so that it will hopefully do the following:

1. elect a new leader (swarm2),
2. start up additional helloworld containers to cover the outage

```
root@swarm1:~# poweroff
Connection to 10.55.0.248 closed by remote host.
Connection to 10.55.0.248 closed.
```

First off, let's list the cluster state.

```
root@swarm2:~# docker node ls
Error response from daemon: rpc error: code = 2 desc = raft: no elected cluster \
leader
```

Uh oh, this was slightly unexpected. After bringing up `swarm1`, I'm seeing that `swarm2` was promoted to a leader. But it's not exactly the fail-over I imagined. While `swarm1` was offline, the ping service only ran as 2/5 and didn't automatically scale on `swarm2` as expected.

```
root@swarm2:~# docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
4i0lko1qdwqp4x1aqwn6o7obh    swarm1    Ready   Active        Reachable
9gyk5t22ngndbwtjof80hpg54 *  swarm2    Ready   Active        Leader
root@swarm2:~# docker service ps helloworld
ID                         NAME              IMAGE   NODE    DESIRED STATE  CURR\
ENT STATE           ERROR
4x0zgeiucsizvmys5orih2bru  helloworld.1      alpine  swarm1  Running        Runn\
ing 3 min
5fxtllouvmd91tmgzoudtt7a4   \_ helloworld.1  alpine  swarm1  Shutdown       Comp\
```

```
lete 3 min
cqvgixx3djhvtiahba971ivr7  helloworld.2      alpine  swarm2  Running        Runn\
ing 21 min
99425nw3r4rf5nd66smjm13f5  helloworld.3      alpine  swarm2  Running        Runn\
ing 21 min
5xzldwvoplqpg1qllg28kh2ef  helloworld.4      alpine  swarm1  Running        Runn\
ing 3 min
1dj3cs7v5ijc93k9yc2p42bhj   \_ helloworld.4  alpine  swarm1  Shutdown       Comp\
lete 3 min
avm36h718yihd5nomy2kzhy7m  helloworld.5      alpine  swarm1  Running        Runn\
ing 3 min
0hy3yzwqzlnee10gat6w2lnp2   \_ helloworld.5  alpine  swarm1  Shutdown       Comp\
lete 3 min
```

So, what went wrong? A bit of reading and I've come up to the following explanation of how Docker uses the RAFT consensus algorithm for leader selection:

> Consensus is fault-tolerant up to the point where quorum is available. If a quorum of nodes is unavailable, it is impossible to process log entries or reason about peer membership. For example, suppose there are only 2 peers: A and B. The quorum size is also 2, meaning both nodes must agree to commit a log entry. If either A or B fails, it is now impossible to reach quorum.

Adding an additional manager to enable fault tolerance

So, if you have three managers, one manager can fail, and the remaining two represent a majority, which can decide which one of the remaining managers will be elected as a leader. I quickly add a `swarm3` node to the swarm. You can retrieve credentials to add nodes by issuing `docker swarm join-token [type]` where type can be either worker or manager.

```
root@swarm2:~# docker swarm join-token manager
To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-0445f42yyhu8z4k3lgxsbnusnd8ws83urf56t02rv1vdh1zqlj-09swsjxd\
z80bfxbc1aed6mack \
    10.55.0.238:2377
```

And we run this command on our `swarm3` machine.

```
root@swarm3:~# docker swarm join \
> --token SWMTKN-1-0445f42yyhu8z4k3lgxsbnusnd8ws83urf56t02rv1vdh1zqlj-09swsjxdz8\
0bfxbc1aed6mack \
> 10.55.0.238:2377
This node joined a swarm as a manager.
root@swarm3:~# docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
4i0lko1qdwqp4x1aqwn6o7obh     swarm1    Ready   Active        Reachable
9gyk5t22ngndbwtjof80hpg54     swarm2    Ready   Active        Leader
b9dyyc08ehtnl62z7e3ll0ih3 *   swarm3    Ready   Active        Reachable
```

Yay! Our `swarm3` is ready. I cleared out the container inventory to start with a clean swarm.

Scaling our service with fault tolerance

I deleted the service with a `docker service rm helloworld`, and cleaned up the containers with a `docker ps -a -q | xargs docker rm`. Now I can start the service again from zero.

```
root@swarm1:~# docker service create --replicas 5 --name helloworld alpine ping \
google.com
5gmrllue1sgdwl1yd5ubl16md
root@swarm1:~# docker service scale helloworld=10
helloworld scaled to 10
root@swarm1:~# docker service ps helloworld
ID            NAME           IMAGE   NODE    DESIRED STATE  CURRENT STATE       \
        ERROR
2hb76h8m7oop  helloworld.1   alpine  swarm1  Running        Running 1 min
5lxefcjclasn  helloworld.2   alpine  swarm3  Running        Running 1 min
95cab7hte5xp  helloworld.3   alpine  swarm2  Running        Running 1 min
a6pcl2fce4hw  helloworld.4   alpine  swarm2  Running        Running 1 min
61rez4j8c5h6  helloworld.5   alpine  swarm1  Running        Running 1 min
2lobeil8sndn  helloworld.6   alpine  swarm1  Running        Running 20 secs
0gieon36unsg  helloworld.7   alpine  swarm1  Running        Running 21 secs
91cdmnxarluy  helloworld.8   alpine  swarm3  Running        Running 21 secs
02x6ppzyseak  helloworld.9   alpine  swarm3  Running        Running 20 secs
4gmn24kjfv7a  helloworld.10  alpine  swarm2  Running        Running 21 secs
root@swarm1:~#
```

And powering off `swarm1`, gives us:

```
root@swarm2:~# docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
4i0lko1qdwqp4x1aqwn6o7obh     swarm1    Ready   Active        Unreachable
9gyk5t22ngndbwtjof80hpg54 *   swarm2    Ready   Active        Leader
b9dyyc08ehtnl62z7e3ll0ih3     swarm3    Ready   Active        Reachable
```

and additional containers have spawned, just as intended:

```
root@swarm2:~# docker service ps helloworld
ID                         NAME              IMAGE   NODE    DESIRED STATE  CURR\
ENT STATE               ERROR
bb8nwud2h75xpvkxouwt8rftm  helloworld.1      alpine  swarm2  Running        Runn\
ing 26 secs
2hb76h8m7oop9pit4jgok2jiu   \_ helloworld.1  alpine  swarm1  Shutdown       Runn\
ing 2 min
5lxefcjclasna9as4oezn34i8  helloworld.2      alpine  swarm3  Running        Runn\
ing 2 min
95cab7hte5xp9e8mfj1tbxms0  helloworld.3      alpine  swarm2  Running        Runn\
ing 2 min
a6pcl2fce4hwnh347gi082sc2  helloworld.4      alpine  swarm2  Running        Runn\
ing 2 min
8n1uonzp2roy608kd6v888y3d  helloworld.5      alpine  swarm3  Running        Runn\
ing 26 secs
61rez4j8c5h6g9jo81xhc32wv   \_ helloworld.5  alpine  swarm1  Shutdown       Runn\
ing 2 min
17czblq9saww4e2wok235kww8  helloworld.6      alpine  swarm2  Running        Runn\
ing 26 secs
2lobeil8sndn0loewrz8n9i4s   \_ helloworld.6  alpine  swarm1  Shutdown       Runn\
ing 1 min
6f3tm5vvhq07kwqt3zu0xr5mi  helloworld.7      alpine  swarm3  Running        Runn\
ing 26 secs
0gieon36unsggqjel48lcax05   \_ helloworld.7  alpine  swarm1  Shutdown       Runn\
ing 1 min
91cdmnxarluy2hc2fejvxnzfg  helloworld.8      alpine  swarm3  Running        Runn\
ing 1 min
02x6ppzyseak8wsdcqcuq545d  helloworld.9      alpine  swarm3  Running        Runn\
ing 1 min
4gmn24kjfv7apioy6t8e5ibl8  helloworld.10     alpine  swarm2  Running        Runn\
ing 1 min
```

Move the services away from a specific node (drain)

With this setup we can tolerate failure of one manager node. But say we wanted a bit more "graceful" procedure of removing containers from one node? We can set the availability to `drain` to empty containers on one node.

```
root@swarm2:~# docker node update --availability drain swarm3
swarm3
root@swarm2:~# docker service ps helloworld | grep swarm3
5lxefcjclasna9as4oezn34i8   \_ helloworld.2  alpine  swarm3  Shutdown       Shut\
down 19 secs
8n1uonzp2roy608kd6v888y3d   \_ helloworld.5  alpine  swarm3  Shutdown       Shut\
down 19 secs
6f3tm5vvhq07kwqt3zu0xr5mi   \_ helloworld.7  alpine  swarm3  Shutdown       Shut\
down 19 secs
91cdmnxarluy2hc2fejvxnzfg   \_ helloworld.8  alpine  swarm3  Shutdown       Shut\
down 19 secs
02x6ppzyseak8wsdcqcuq545d   \_ helloworld.9  alpine  swarm3  Shutdown       Shut\
down 19 secs
root@swarm2:~# docker service ps helloworld | grep swarm2 | wc -l
10
```

All the containers on `swarm3` shut down, and started up on the remaining node, `swarm2`. Let's scale down the example to only one instance.

```
root@swarm2:~# docker service scale helloworld=1
helloworld scaled to 1
root@swarm2:~# docker service ps helloworld | grep swarm2 | grep -v Shutdown
17czblq9saww4e2wok235kww8  helloworld.6      alpine  swarm2  Running        Runn\
ing 7 min
```

Cleaning up the containers is still very much in the domain of the sysadmin. I started up `swarm1`, and scaled our service to 20 instances.

```
root@swarm2:~# docker service scale helloworld=20
helloworld scaled to 20
root@swarm2:~# docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
5gmrllue1sgd  helloworld  2/20      alpine  ping google.com
root@swarm2:~# docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
5gmrllue1sgd  helloworld  10/20     alpine  ping google.com
root@swarm2:~# docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
5gmrllue1sgd  helloworld  16/20     alpine  ping google.com
root@swarm2:~# docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
5gmrllue1sgd  helloworld  20/20     alpine  ping google.com
```

As you can see here, it does take some time for the instances to start up. Let's see how it distributed.

```
root@swarm3:~# docker service ps -f "desired-state=running" \
               helloworld | grep swarm | awk '{print $4}' | sort | uniq -c
     10 swarm1
     10 swarm2
```

Enabling and scaling to a new node

As we put `swarm3` into `drain` availability, we don't have any instances running on it. Let's fix that very quickly by putting it back into `active` availability mode.

```
root@swarm3:~# docker node update --availability active swarm3
swarm3
```

As the already running service will stay the same, we need to scale our service to populate `swarm3`.

```
root@swarm3:~# docker service scale helloworld=30
helloworld scaled to 30
root@swarm3:~# docker service ps -f "desired-state=running" \
               helloworld | grep swarm | awk '{print $4}' | sort | uniq -c
     10 swarm1
     10 swarm2
     10 swarm3
```

It takes a bit of getting used to, but docker service is a powerful way to scale out your microservices. It might be slightly more tricky when it comes to data volumes (mounts), but that's the subject of another post.

When you're provisioning swarm managers, you need a majority to resolve failures gracefully. That means you should have an odd number of managers, where N > 2. A cluster of N managers is able to tolerate failure of ((N-1)/2) nodes, for example 3 managers = 1 failed node, 5 managers = 2 failed nodes, 7 managers = 3 failed nodes and so on.

A worker in comparison doesn't replicate the manager state, and you can't start or query services from a worker. You should only do that from any of the manager nodes - the commands will be ran on the current leader node.

Scaling our service with Docker Swarm

As we now have a docker swarm set up, we will use it to orchestrate a deployment and a rolling upgrade of our service. The image of the application which you want to deploy should be available on all swarm nodes, either via your custom registry, or docker hub. For the purposes of this chapter, I've pushed the 'gotwitter' image to Docker hub. This way, the image is available as `titpetric/gotwitter` - and in the case of production it would be available from your docker registry.

Disk space

According to titpetric/gotwitter docker hub page[83] the image is about 5MB in size at the time of writing. Keep in mind, this is the compressed size, the actual size will be reported using `docker images | grep gotwitter`:

```
# docker images | grep gotwitter
titpetric/gotwitter    latest    65d327973482     8 secs   12 MB
```

Why is disk size important? As you've learned by now, the sizes of images range from a few megabytes up to and over a few gigabytes. If you want to scale your service on your hardware, you will need to plan ahead and provide the necessary disk space to scale your services accordingly.

The swarm nodes which I created, I provisioned with 160GB of disk space. To make it simpler, I'll plan to use about 120GB for containers - with our app container here, this would give us the capacity of running about 10 thousand containers on one node. If the container was around 1GB, we could only run about 120 of them. Of course, in our case, disk is not the limiting factor, but RAM usage might be.

RAM usage

I've provisioned 8GB of ram to each swarm manager. Inspecting the gotwitter app, we can see that it uses less than 10 MB of memory. This number will likely be higher when your application will be running for a while so keep that in mind, when provisioning containers.

```
# docker stats gotwitter
CONTAINER   MEM USAGE / LIMIT       MEM %    NET I/O         BLOCK I/O
gotwitter   8.559 MiB / 1.886 GiB   0.44%    648 B / 648 B   8.262 MB / 0 B
```

> I started the container off the cluster. By default, if the container is not started with any memory limits, it may take up to the full amount available on the system (which in this case is about 2GB).

[83] https://hub.docker.com/r/titpetric/gotwitter/tags/

A basic back-of-the-envelope calculation[84] gives us a reasonable guess that we can run about 800 containers on a single swarm node, with the above memory consumption. Given that we have 3 nodes, the guess is that we could run about 2000 containers. Generally this is pretty bad guessing, since we don't know what kind of memory consumption is required outside of the running container, nor what kind of memory usage we can expect of our application after it has been running for a while.

Creating a multi-host network

At this point, it would be worth mentioning that the `bridge` network we created at the beginning of the book is contained to a single host. In order for containers to be discoverable between hosts in a docker swarm, we will need to create an `overlay` network on one of the manager nodes (all, in our case, as we don't have any workers in the swarm).

```
# docker network create \
  --driver overlay \
  --subnet 10.0.0.0/20 \
  --attachable \
  party-swarm
```

> Note: I am using the `/20` CIDR notation here. The usual `/24` gives you 256 IPs (2^8), while '/20' gives you 4096 IPs (2^{12}). One of the IPs (10.0.9.1) is used by the Docker host to provide routing. Each service you create in swarm mode gets a Virtual IP, and each container that's created by the service gets it's own IP. So, if you define 10 services, each running 2 containers, 1 + 10 + (10 * 2) = 31 IPs will be used. This is important when scaling your service, as it limits you to how many instances you can have on the network.

When creating a service, we can now use `--network party-swarm`. Docker swarm will extend the network to all the manager nodes where the containers will be running. Using the `--attachable` switch when we created the swarm network enables non-managed containers (i.e., `docker run`) to attach to the network as well.

[84] https://en.wikipedia.org/wiki/Back-of-the-envelope_calculation

```
# docker run --rm --net=party-swarm alpine ifconfig eth0 | grep "inet addr"
-- swarm1 --
          inet addr:10.0.9.2  Bcast:0.0.0.0  Mask:255.255.255.0
-- swarm2 --
          inet addr:10.0.9.2  Bcast:0.0.0.0  Mask:255.255.255.0
-- swarm3 --
          inet addr:10.0.9.2  Bcast:0.0.0.0  Mask:255.255.255.0
```

The jig of the thing is that when you run containers in sequence, you will re-use the first available IP. This is why when we run the alpine container on each of the swarm nodes, it will give the same IP as the previous command already exited. As a test, we'll create a Redis service in the `party-swarm` network.

```
root@swarm1:~# docker service create --replicas 1 --network party-swarm --hostna\
me redis --name redis -p 6379:6379 redis:3.2-alpine
3iy1z0hc9gsb6l7avsv8a2903
```

We can now inspect the redis container running on the same host. You should get the name of the container by running `docker ps`.

```
root@swarm1:~# docker exec -it  redis.1.kcd5v3xxiwro830tn8oab6a4d ifconfig eth2 \
| grep "inet addr"
          inet addr:10.0.0.3  Bcast:0.0.0.0  Mask:255.255.255.0
```

So, our redis instance is up and running on 10.0.0.3. By creating a container on a different host, we can verify that we get an additional IP in the network, and that we have connectivity to the redis service.

```
root@swarm3:~# docker run --rm --net=party-swarm alpine ifconfig eth0 | grep "in\
et addr"
          inet addr:10.0.0.4  Bcast:0.0.0.0  Mask:255.255.240.0
root@swarm3:~# docker run -it --rm --net=party-swarm alpine sh -c "echo quit | n\
c redis 6379"
+OK
root@swarm3:~# docker run --rm --net=party-swarm alpine ping -c1 redis
PING redis (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: seq=0 ttl=64 time=0.098 ms
```

Note that the `redis` name resolves into the Virtual IP for the Redis service as was mentioned above. If we scale the redis service to more instances, the VIP will provide connectivity to all of them. Applications connect to the VIP, and docker swarm handles routing to individual containers active within the service. This is also used when doing rolling upgrades.

> By default, when you create a service attached to a network, the swarm assigns the service a VIP. The VIP maps to a DNS alias based upon the service name. Containers on the network share DNS mappings for the service via gossip so any container on the network can access the service via its service name.
>
> For more information you can consult the official Docker documentation[85]

In this case, `10.0.0.2` is the VIP (Virtual IP). You can inspect the service to view the virtual IP. For example:

```
root@swarm3:~# docker service inspect --format='{{json .Endpoint.VirtualIPs}}' r\
edis
[{"NetworkID":"kc8yy8tc62qrmq4xhqu4g74yl","Addr":"10.255.0.5/16"},{"NetworkID":"\
q3l72085za9s20fwxfix83i7k","Addr":"10.0.0.2/20"}]
```

If you will note, we actually have two networks attached to the redis service. Because I added `-p 6379:6379` docker also added the container to the "ingress" network. This is provided by docker as a way to forward traffic from the host to individual service VIPs. You can verify this by getting additional information from `docker network ls`:

```
root@swarm3:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
97ef30f43a1d        bridge              bridge              local
7173e4240ad9        docker_gwbridge     bridge              local
6b5343cc52b5        host                host                local
kc8yy8tc62qr        ingress             overlay             swarm
c59c625e2ffa        none                null                local
q3l72085za9s        party-swarm         overlay             swarm
```

So, our network in practice looks like:

```
10.0.0.1 the docker host we're running on (swarm1, swarm2 or swarm3 respectively)
10.0.0.2 the Virtual IP for the `redis` service, routes to individual containers
10.0.0.3 the IP for a `redis` container
10.0.0.4 our Alpine container which we ran
```

[85] https://docs.docker.com/engine/swarm/networking/

Sysctl tunable settings

Running a 1000 containers is not a small feat. In fact, in a single host, you're somewhat limited to running slightly less than 1024 containers because of a variety of reasons. You can however make it slightly easier on you, to tune parts of your system, to make it less painful to run such a high number of containers on one server. Honestly, you shouldn't run so many mainly because I suspect that you don't have a server with 1000 cores available.

When running applications in Linux, system parameters can be tuned with `sysctl`:

> sysctl is a feature of some Unix-like operating systems that reads and modifies the attributes of the system kernel such as its version number, maximum limits, and security settings. It is available both as a system call for compiled programs, and an administrator command for interactive use and scripting.

Various Linux subsystems, especially in regards to networking, have many exposed configuration flags that are managed by sysctl. The defaults of these flags are more suited for laptops, than comparatively servers that would run thousands of processes.

When I first tried to run 1000 containers in a service, I hit a limitation that resulted in the docker swarm becoming unresponsive, with occasional connectivity interruptions. When inspecting this, I found many messages like the following one in output of `dmesg`.

```
neighbour: arp_cache: neighbor table overflow!
```

After researching the cause on google, I did find suggestions of which sysctl values I should increase in order to increase ARP caches, the underlying reason for the error:

> *ARP cache is overflowing.* Most possible reason - too much traffic on the network (generated by some application, by some hosts, or by related factors).

A bit of digging around lead me to some sysctl tunable settings which can be used to work around the problem; basically by increasing the thresholds where ARP cache collection occurs.

Issue this on every docker swarm host:

```
sysctl -w net.ipv4.neigh.default.gc_thresh1=8096
sysctl -w net.ipv4.neigh.default.gc_thresh2=8096
sysctl -w net.ipv4.neigh.default.gc_thresh3=8096
```

Source: docker/libnetwork GitHub issue tracker #1522[86]. The reason why I chose this value is that I needed a sane default, that would not be hit. Generally the recommendation is that you should multiply all the `gc_thresh*` values by 2 until your system will behave nicely. As is the case most of the time, the sysctl tunable settings are very poorly documented, so figuring out what kind of impact they have in terms of memory use or other pitfalls is possible only in the rarest occasions.

And there are other sysctl tunable settings which might be useful for you:

[86] https://github.com/docker/libnetwork/issues/1522

```
# Have a larger connection range available
net.ipv4.ip_local_port_range=1024 65000

# Reuse closed sockets faster
net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_fin_timeout=15

# The maximum number of "backlogged sockets".  Default is 128.
net.core.somaxconn=4096
net.core.netdev_max_backlog=4096

# 16MB per socket - which sounds like a lot, but will virtually never consume th\
at much.
net.core.rmem_max=16777216
net.core.wmem_max=16777216

# Various network tunables
net.ipv4.tcp_max_syn_backlog=20480
net.ipv4.tcp_max_tw_buckets=400000
net.ipv4.tcp_no_metrics_save=1
net.ipv4.tcp_rmem=4096 87380 16777216
net.ipv4.tcp_syn_retries=2
net.ipv4.tcp_synack_retries=2
net.ipv4.tcp_wmem=4096 65536 16777216
#vm.min_free_kbytes=65536

# Connection tracking to prevent dropped connections (usually issue on LBs)
net.netfilter.nf_conntrack_max=262144
net.ipv4.netfilter.ip_conntrack_generic_timeout=120
net.netfilter.nf_conntrack_tcp_timeout_established=86400

# ARP cache settings for a highly loaded docker swarm
net.ipv4.neigh.default.gc_thresh1=8096
net.ipv4.neigh.default.gc_thresh2=8096
net.ipv4.neigh.default.gc_thresh3=8096
```

These sysctl settings deal mainly with how many connections can be open at a given time, which ports are used for the connections, how fast connections are recycled,...

Rolling updates for our service

Let's create our service and start with a relatively small number:

```
root@swarm3:~# docker service create -e PORT=80 \
    --replicas 10 \
    --network party-swarm \
    --update-parallelism 5 \
    --name gotwitter \
    -p 80:80 titpetric/gotwitter
shmnmoz80hz772e4x2ksylyr9
```

When the docker service is created, docker swarm always pulls to the latest image which you are referencing. If you want to do a rolling upgrade, it can be as simple as this:

```
root@swarm3:~# docker service update --image titpetric/gotwitter gotwitter
gotwitter
root@swarm3:~# docker service ls
ID            NAME       MODE        REPLICAS  IMAGE
j204e4p7q70q  gotwitter  replicated  5/10      titpetric/gotwitter:latest
syhtbxnz8zhh  redis      replicated  1/1       redis:3.2-alpine
```

As you can see, 5 instances are updated at a time, based on the –update-parallelism switch when we created our service. The speed of the upgrade is limited by your health check interval if you have one. My health check is set for 60 seconds, which means that the service needed 2 minutes to upgrade to the new settings.

`--update-parallelism [uint]` - Maximum number of tasks updated simultaneously (0 to update all at once) (default 1)

In fact, a breakdown of what happened is:

1. Swarm shut down [update-parallelism] instances,
2. Swarm started new [update-parallelism] instances,
3. Swarm waited for health checks to pass,
4. Swarm started routing traffic to new instances,
5. Back to step 1 until there are some old containers left

After some time, when all the health check have passed and all the containers are up and running, you can inspect what happened on a single host with a `docker ps -a`. I'm shortening the output a bit here.

```
IMAGE                    CREATED     STATUS                  NAMES
gotwitter@sha256:6b8b    14 min      Up 14 min (healthy)     gotwitter.5.olgv34
gotwitter@sha256:6b8b    14 min      Up 14 min (healthy)     gotwitter.7.copxm8
gotwitter@sha256:6b8b    14 min      Up 13 min (healthy)     gotwitter.4.yjmsk1
gotwitter@sha256:6b8b    14 min      Up 14 min (healthy)     gotwitter.2.leh34n
gotwitter@sha256:6b8b    14 min      Up 14 min (healthy)     gotwitter.3.l1588x
gotwitter@sha256:dee7    15 min      Exited (2) 14 min       gotwitter.7.kyhbaq
gotwitter@sha256:dee7    15 min      Exited (2) 14 min       gotwitter.4.we7dw1
gotwitter@sha256:dee7    15 min      Exited (2) 12 min       gotwitter.9.bdztk0
gotwitter@sha256:dee7    15 min      Exited (2) 14 min       gotwitter.3.kkxen4
```

As you can see, it took 2 minutes to perform the rolling upgrade. The image ID changed, because a newer version of the gotwitter image was available on the docker registry.

> Note: This is actually, a pretty bad way to do a rolling update. How do you know which image you will end up with? Can you do a rollback? This is a good point to remind you that you should tag your builds either with git commit hashes or timestamps, or something. With tags, you have explicit control which image should be started on your infrastructure.

Scaling to a 1000 containers!

With our sysctl tunable settings in place, and our service created - we have all the prerequisites to scale our service to a 1000 containers.

At this point it would be very useful to turn off health checks, as running them will slow us down very much - imagine a 1000 containers running at least two processes for a health check every minute. Each health check runs `sh -c [healthcheck]`, which ends up being at least two processes.

```
root@swarm3:~# docker service update --no-healthcheck gotwitter
gotwitter
```

This will restart all the existing containers, in order to turn off health checks. And now we're ready to scale up!

> If you're running docker 1.13 (as I was), you might find ourself in the situation, where the published network port becomes unpublished after turning off health checks. I documented this issue in the Docker GitHub[87] and also provided a work-around for you:
>
> `docker service update --health-cmd "exit 0" gotwitter`
>
> While this still has significant overhead (of one 'sh' processes anyway), it should be reasonable enough to reach an active 1000/1000 containers for our service.

Before we scale, let's just inspect our service, to see that it completed successfully:

```
# docker service inspect gotwitter --pretty
ID:             j204e4p7q70q2nnqneg9k3y9n
Name:           gotwitter
Service Mode:   Replicated
 Replicas:      10
UpdateStatus:
 State:         completed
 Started:       2 minutes
 Completed:     9 seconds
 Message:       update completed
...
```

So let's go from 10 to 1000. Issue the following command to scale the gotwitter service to 1000 active containers.

[87] https://github.com/docker/docker/issues/31342

```
root@swarm3:~# docker service scale gotwitter=1000
gotwitter scaled to 1000
```

This will take some time - on my system it took somewhere along the lines of 12 minutes; I'm monitoring progress with `docker service ls` to estimate how much time it will take to scale up to the requested number of replicas. Of course, spawning such a large number of containers is an I/O and CPU intensive process.

```
ID            NAME       MODE        REPLICAS   IMAGE
j204e4p7q70q  gotwitter  replicated  1000/1000  titpetric/gotwitter:latest
syhtbxnz8zhh  redis      replicated  1/1        redis:3.2-alpine
```

Here we have a 1000 container cluster, running on 3 hosts. Yay!

A word on health checks

If you're running health checks on such a host to container ratio, you will get some lag even with the most simple health checks. If your application returns some HTTP response in 1 ms, you shouldn't expect that a timeout of one or two seconds is reasonable, even 10 seconds is a far cry from real world conditions in such a scenario.

Keep in mind, I'm running this on actual *server* hardware. If you want to run so many containers on that PC you keep under the desk, perhaps your laptop, or some more esoteric platform that runs docker but is low on raw performance - your results will be much worse.

In case your service doesn't get up to `1000/1000`, this is a sign that either you managed to consume your available resources (disk, ram) and new containers can't be started, or just perhaps that your health check is failing.

My first suggestion would be to increase your health check interval to at least 60s. I'm running my health check every 60 seconds, with a 60 second timeout. While this may not give the best response time when a service starts failing, it's reasonable enough that it didn't give false-positive reports of a failing service in our case.

More about health checks will be said in the next chapter.

Inspecting final container state

It seems that my estimates about memory and disk usage were correct. Keep in mind, this is still relatively "cold" start, once the application get's hot it will use more memory, and scaling the number of containers to use all the RAM does not make sense.

```
root@swarm1:~# free -h
              total        used        free      shared  buff/cache   available
Mem:           7.8G        3.9G        1.6G         62M        2.3G        3.6G
Swap:          2.0G         18M        2.0G
root@swarm1:~# df /dev/sda1 -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       156G  2.2G  146G   2% /
```

Scaling beyond a 1000 containers

The most viable option to go beyond 1000 containers per host is to use a different kind of virtual networking driver, like `openvswitch`. With it, you can bypass the limitation of the default network driver, which only allows allocations of up to 1024 IPs.

This limitation is declared in the Linux kernel, so another option is to modify and recompile the kernel. You can consult this GitHub issue[88] for more information about these limitations and workarounds.

[88] https://github.com/docker/docker/issues/1320#issuecomment-22346764

IX. Disposability - Maximize robustness with fast startup and graceful shutdown

> The twelve-factor appâ€™s processes are disposable, meaning they can be started or stopped at a momentâ€™s notice. This facilitates fast elastic scaling, rapid deployment of code or configuration changes, and robustness of production deploys.

Disposing of containers

As we still have our 1000 container service running from the previous chapter, there is a basic test which can be done which effectively is the same as a container disappearing. Since we follow 12FA guidelines, this means that:

1. There should be no data loss as there is no data stored by the container,
2. A new container can be started in the place of the old one

Very trivially, I want to kill off the last three ‘gotwitter.*’ containers on one of the hosts, and then monitor the output of `docker service ls`.

```
# docker ps -a | grep 'gotwitter\.[0-9]' | head -n 3 | awk '{print $1}' | xargs \
docker rm -f
1e23a65d0cd5
73cbb39fcd2e
48bd977b284a
# docker service ls
ID            NAME       MODE        REPLICAS  IMAGE
j204e4p7q70q  gotwitter  replicated  997/1000  titpetric/gotwitter:latest
syhtbxnz8zhh  redis      replicated  1/1       redis:3.2-alpine
```

The failures are detected by docker right away, but due to the health checks, it will take a minute before the new containers are marked as healthy.

```
ID            NAME       MODE        REPLICAS   IMAGE
j204e4p7q70q  gotwitter  replicated  1000/1000  titpetric/gotwitter:latest
syhtbxnz8zhh  redis      replicated  1/1        redis:3.2-alpine
```

In terms of testing out resilience of individual container services, we can envision a system in which we add a 'disposable' label to our service which follows 12FA guidelines. It is then trivial to list the services which are disposable:

```
root@swarm3:~# docker service create \
      -e PORT=81 \
      --replicas 3 \
      --network party-swarm \
      -l role=disposable \
      --name gotwitter-disp \
      -p 81:81 titpetric/gotwitter
nfnaya4wqj67ntfqzutdyuq7h
```

After creating a service with a label name `role` with the value `disposable`, we can filter services by this role.

```
root@swarm3:~# docker service ls --filter=label=role="disposable"
ID            NAME            MODE        REPLICAS  IMAGE
nfnaya4wqj67  gotwitter-disp  replicated  3/3       titpetric/gotwitter:latest
```

From then it just takes a simple `docker service ps [service-name]` to list the running containers on a service, and choose which ones you would like to kill off.

```
root@swarm3:~# docker service ps gotwitter-disp
ID            NAME              IMAGE                         NODE    DESIRED STAT\
E  CURRENT STATE           ERROR  PORTS
xz3x5js48wkm  gotwitter-disp.1  titpetric/gotwitter:latest  swarm2  Running     \
   Running 14 seconds ago
huvt13uq2vt3  gotwitter-disp.2  titpetric/gotwitter:latest  swarm1  Running     \
   Running 10 seconds ago
qriblhwebpam  gotwitter-disp.3  titpetric/gotwitter:latest  swarm3  Running     \
   Running 7 seconds ago
```

We can even filter the containers by the current node:

```
root@swarm3:~# docker service ps gotwitter-disp -f node=$(uname -n)
ID            NAME              IMAGE                         NODE    DESIRED STAT\
E  CURRENT STATE          ERROR  PORTS
qriblhwebpam  gotwitter-disp.3  titpetric/gotwitter:latest  swarm3  Running     \
   Running 3 minutes ago
```

All just to kill off one of them:

```
root@swarm3:~# docker service ps gotwitter-disp -f node=$(uname -n) --no-trunc |\
 grep Run | awk '{print $2 "." $1}' | tail -n 1 | xargs --no-run-if-empty docker\
 rm -f
gotwitter-disp.3.qriblhwebpambrytekalr0c5i
root@swarm3:~# docker service ps gotwitter-disp
ID            NAME                  IMAGE                         NODE    DESIRED \
STATE  CURRENT STATE            ERROR                         PORTS
xz3x5js48wkm  gotwitter-disp.1      titpetric/gotwitter:latest  swarm2  Running \
       Running 6 minutes ago
huvt13uq2vt3  gotwitter-disp.2      titpetric/gotwitter:latest  swarm1  Running \
       Running 6 minutes ago
vfznagkyr1pb  gotwitter-disp.3      titpetric/gotwitter:latest  swarm3  Running \
       Starting 13 seconds ago
qriblhwebpam   \_ gotwitter-disp.3  titpetric/gotwitter:latest  swarm3  Shutdown\
       Failed 18 seconds ago    "task: non-zero exit (137)"
root@swarm3:~# docker service ls
ID            NAME            MODE        REPLICAS  IMAGE
69fa7gqlzbhf  gotwitter-disp  replicated  2/3       titpetric/gotwitter:latest
```

This method of development/testing is employed at Netflix, in the form of their Chaos Monkey product[89]. Chaos Monkey is their attempt to weed out design flaws in their systems, by actively testing their systems to tolerate failure.

[89] https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

Docker Chaos Monkey

I slightly extended the principles of disposing containers and packaged it into a solution which you can use outside of this book. I created "docker-chaos-monkey", which is available on GitHub[90].

It aims to provide a mechanism to kill a single container from a service, which is marked as disposable with a label like `role=disposable`. In addition to that, it has a few rules that make running it a bit more safe:

- You need more than 1 container in a service to kill it off,
- The service must be fully replicated (will not kill if service is degraded),
- Only kills a container if it's running on the current node

[90] https://github.com/titpetric/docker-chaos-monkey

```
root@swarm1:~/docker-chaos-monkey# ./chaos.sh
--------------------------------------------------------------------------------
| Running this script will kill off 1 docker image with label: role=disposable
| You have 5 seconds to change your mind and CTRL+C out of this.
--------------------------------------------------------------------------------
mmqe8ua6xcdp  gotwitter   replicated  5/5       titpetric/gotwitter:latest
zlca3iOz6hln  gotwitter2  replicated  1/1       titpetric/gotwitter:latest
--------------------------------------------------------------------------------

mmqe8ua6xcdp gotwitter: swarm1
removing a container
> gotwitter.5.uy208kxdblek1m7rrk37by029

zlca3iOz6hln gotwitter2: service has only one running container - skipping

--------------------------------------------------------------------------------
Final service status:
ID            NAME        MODE        REPLICAS  IMAGE
mmqe8ua6xcdp  gotwitter   replicated  4/5       titpetric/gotwitter:latest
syhtbxnz8zhh  redis       replicated  1/1       redis:3.2-alpine
zlca3iOz6hln  gotwitter2  replicated  1/1       titpetric/gotwitter:latest
root@swarm1:~/docker-chaos-monkey#
root@swarm1:~/docker-chaos-monkey#
root@swarm1:~/docker-chaos-monkey# SKIP_WARNING=1 ./chaos.sh
mmqe8ua6xcdp gotwitter: service is degraded 4/5 - skipping
zlca3iOz6hln gotwitter2: service has only one running container - skipping

--------------------------------------------------------------------------------
Final service status:
ID            NAME        MODE        REPLICAS  IMAGE
mmqe8ua6xcdp  gotwitter   replicated  4/5       titpetric/gotwitter:latest
syhtbxnz8zhh  redis       replicated  1/1       redis:3.2-alpine
zlca3iOz6hln  gotwitter2  replicated  1/1       titpetric/gotwitter:latest
root@swarm1:~/docker-chaos-monkey#
```

Running Docker Chaos Monkey

With it, you're on your way to adapt your application and services to better tolerate failure and continue working if a container goes offline.

Health checks

Depending on how many times you're building your applications, your system will need to handle the startup of new application instances and phasing out old ones. It is very important that start up time for new services is fast, so any possible glitches can be minimal. Docker swarm handles rolling upgrades for us, but only recently docker added support for running health checks for the containers you run. You will need at least Docker 1.12 for this functionality.

Writing `HEALTHCHECK` instructions enables you to inspect the state of your docker container, giving docker additional information if your application is ready for processing requests. For example, starting up a MySQL instance might take more than 10-20 seconds, before the container is ready to serve queries to clients. In some cases, even web servers like `nginx` might take some time before they are ready, and the same goes for your Go application.

A simple way to test reachability of your application might be to issue a HTTP request against the service port of your application. This is another reason why configuration needs to come from the environment - health checks will not be aware of any command line options which you may pass to your application, and if you're using `.env` files with godotenv, you'll need to implement reading from this file with something as simple as a bash script. Or just use the environment, as you should!

A `curl` check should be enough to know if your application is ready to serve requests. With the option `-f` (short for `--fail`), curl will produce a non-zero exit code which will tell docker that the container is not yet ready.

```
1 HEALTHCHECK --interval=60s --timeout=60s CMD curl -f http://localhost:$PORT/api/\
2 health || exit 1
```

There's a few things here that we should pay attention to:

Health check options

Health check doesn't have many options available:

`--interval` - this option specifies the time interval how often your health check will run. It also specifies when your first health check will run. I suspect that Docker will eventually improve this option to implement some kind of back-off as soon as a health check passes. The idea is that you want your service to be marked healthy as soon as possible, but you might not care as much after that and would like to back off of high frequency checks.

The default interval is 30 seconds.

`--timeout` - this option specifies the timeout where it should be obvious that there's something wrong with your application if the health check doesn't complete within this time frame. If the check takes longer than this time, the health check will be considered failed.

The default interval is 30 seconds.

`--retries=N` - this option specifies after how many failed health checks should fail before the container is marked as `unhealthy`.

The default duration is 3 times. This seems good for us, but you should pay attention when writing health checks for software that would take a good while to spin up.

The health checks can be controlled from `docker service create` or `docker service update`, so theoretically you may turn them off, or make them run at different intervals than the ones set in the Dockerfile. There was more about this in the previous chapter, where we scaled our service to 1000 running containers.

Various

As I mentioned, we need to know the PORT on which our application will be running. This value is passed via environment variables. Of course, a default value should be provided in case there's no configuration coming from the environment:

```
1 ENV PORT=3000
```

This statement in the Dockerfile will declare the environment variable `PORT`, and set it to 3000 if the environment variable didn't exist beforehand. It will keep it's value if it exists. The PORT variable is then explicitly used in the command that follows it.

Finally, the final `|| exit 1` ensures that if curl returns an unexpected non-zero exit code, that the exit code returned to docker will be the expected `1`, indicating a failed test. This would also be true if for example `curl` wouldn't be installed in the system - also triggering a failed test.

X. Dev/prod parity - Keep development, staging, and production as similar as possible

Docker images are everything you need to run an application. This application might need external data which you pass via the environment, the filesystem, or backing services. Depending on which environment you run a container, any of these external sources might change. But the one thing that doesn't is the docker image - the same exact image can be used in all environments.

Why does this rule exist? Docker pretty much solves this rule immediately, but it's good to review why this rule exists in the first place.

It worked on my machine

Of course, the most common excuse is usually “It worked on my machine!”. But your machine has software installed as you saw fit, it takes great discipline to use Vagrant or some other tool to manage your development environment.

Well, it did. Docker images can contain any software package you think of, and all the steps of making the docker image are documented. It’s automation in the most creepy sense - it’s literally everything you do to set up a virtual machine, except that the result at the end is that you can upgrade it anytime while keeping older versions, and “It worked on my machine!” excuse can never again be uttered.

Upgrades and testing

The way upgrades usually work with virtual machines / VPS is that people set up new environments, install the software that's needed, and then see which software breaks. The better projects have at least unit tests that scream horribly when something goes wrong. But at some point - even infrastructure needs to be tested, not just software.

A very common pattern that emerged when building docker images is testing the capabilities of the software that it comes with. These tests go beyond just version restrictions (of course, knowing that PHP doesn't jump from 5.6 to 7.0 is a good thing).

Maybe you need to know that LUA is installed in your container, and that it works as expected (and doesn't crash for some reason like missing libraries).

```
#!/bin/bash
set -e
TEST=$(echo 'print("hello")' | lua -)
if [ "$TEST" == "hello" ]; then
  echo "Lua check passed OK"
else
  echo "Lua check failed, unexpected output: ${TEST}"
  exit 1
fi
```

Or maybe you need to test really specific NGINX configuration. I'm using the LUAJIT module in NGINX usually, and some distributions (*alpine*) don't compile nginx with this capability. Good to know that you don't get the same nginx in alpine. Because a test is failing if you try.

```
#!/bin/bash
set -e
# name of script without '.sh' extension
CONFIG_FOLDER=$(realpath -s $0 | cut -d'.' -f1)
CONFIG="$CONFIG_FOLDER/nginx.conf"

echo "NGINX Testing" $CONFIG

# test configs that need to pass without errors
nginx -c ${CONFIG} -t
```

Maybe you're using something much more fragile, like ImageMagick, LUAJIT, FFI and a patched project from GitHub which gives LUA some image processing capabilities. Believe me, if this doesn't work you may take days replacing this functionality with other software. Good to have tests for your critical components.

```
#!/bin/bash
set -e
git clone https://github.com/titpetric/magick.git
cd magick
luarocks make
# luarocks install magick

TEST_FOLDER=$(realpath -s $0 | cut -d'.' -f1)
cd $TEST_FOLDER
echo -n "LUA magick test: "
luajit test.lua
if [ ! -f "output.png" ]; then
  echo "LUA test failed, no output.png (magick)"
  ls -la .
  exit 1
fi
echo "PASS"
```

There was a great podcast a while ago on software engineering daily[91]. I'm going to paraphrase here a bit, this is not a literal quote:

> At Uber, there are many technology stacks in production, some from people who are no longer with the company. There's no complete overview of what runs where and who is responsible for it. But to be fair, Uber is not "brownfield" software, the critical path is well tested and understood.

So, the idea is - if a feature is critical for you, it should be well tested so you can ensure that it stays functional between rebuilds of your docker images. Tests like the above ensure that at leas the tested functionality is available and working as expected. Since the tests are performed on build of the docker image, problems will be detected before they can have any impact on production environments.

[91] https://softwareengineeringdaily.com/

Many moving parts

When we think about production or development environments for a simple app, it's very easy to say something like "Oh just run this and you're good to go". In reality, a system that's not in the beginning stages will have things like load balancers, web servers, database servers, file servers, caching servers, api servers, monitoring, management, and so on. When we're talking about having development and production be as similar as possible, this is also one aspect of it - we're not talking about just one container, but how all the containers interact with each-other.

Enter docker-compose[92].

> Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your applicationâ€™s services. Then, using a single command, you create and start all the services from your configuration. To learn more about all the features of Compose see the list of features.
>
> Compose is great for development, testing, and staging environments, as well as CI workflows.

We can use Docker compose to start up the web server, the database server and your application if needed. When it comes to developing with Go applications, it's not very practical to wait very long so that your Go code will compile and then be packaged into a container. But if you're developing a Vue front-end to the APIs that are exposed by a Go service, then you will most likely not even need to restart the containers once you start them. As with standard `docker run` commands, compose supports volume mounts, which means you can edit your code directly from a git checkout on your development machine, and the containers can access the files over a volume mount.

You'll note that the description of compose explicitly skips mentioning `production` environments. Compose is a wrapper around docker run which works on a single host. Even with some swarm support, it just doesn't provide the granularity that's required when dealing with a production system. That being said, for setting up a simple development environment it's great - all the containers in such environments are throw-away.

[92] https://docs.docker.com/compose/

XI. Logs - Treat logs as event streams

The 12 Factor guidelines say:

> A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the appâ€™s behaviour.

To make it a bit more obvious, this rule concerns itself with two things:

First concern: When you run a docker container in the foreground, you should be able to inspect the output as it happens. This may mean anything from access logs, to less verbose outputs like specific action audit logs.

```
# docker run -it --rm debian:jessie bash -c 'for i in {1..5}; do echo $i; done'
1
2
3
4
5
```

This concern enables you to see the output of your program in real time, in production and development. When you're running your program as a daemon, you issue a `docker logs [container]` command to inspect all output, and `docker logs -f [container]` to follow the output as your application generates it. This is the equivalent of using `tail -f [file]`, any new output will be printed immediately so you can see it.

Second concern: if you're managing the routing or storage in your app, this would be subject to failure, much as the rest of your app. Your application should have a sense of single responsibility, and if it has to collect, store and manage logs in addition to it's primary purpose, it may be less reliable overall, because a failure in one part of the application might mean disruption over all parts.

When we're talking about 12 Factor applications we must realize that they are not only Go programs, and in some cases proper error handling is next to impossible. Unexpected crashes occur, and having log streaming in place means that we can preserve some detail which would be lost if you have logging implemented in the application.

A note on backing services

Backing services may enable you to store some of this real-time stream in a database or service that it specialized to handle large amounts of data. The reason for the division of logs between a backing service and the stdout is one of verbosity. The standard output stream should deal with system operation logs, and not necessarily the access log for your complete service. You can use the environment to specify if your application should output the access log to the stdout (for development), or if it should be stored in a separate backing service.

If an error occurs and your application exists - this must be visible in the stdout/stderr event streams. These events should be forwarded to a specific service that deals with operations.

Meanwhile, thousands or millions or more API requests to your service might not be suited for the same stream output any more. While you surely can output those to the stdout, it will become harder finding exceptions in such a log stream, because of the high verbosity. Structured logging aims to fix some of the pain associated with this.

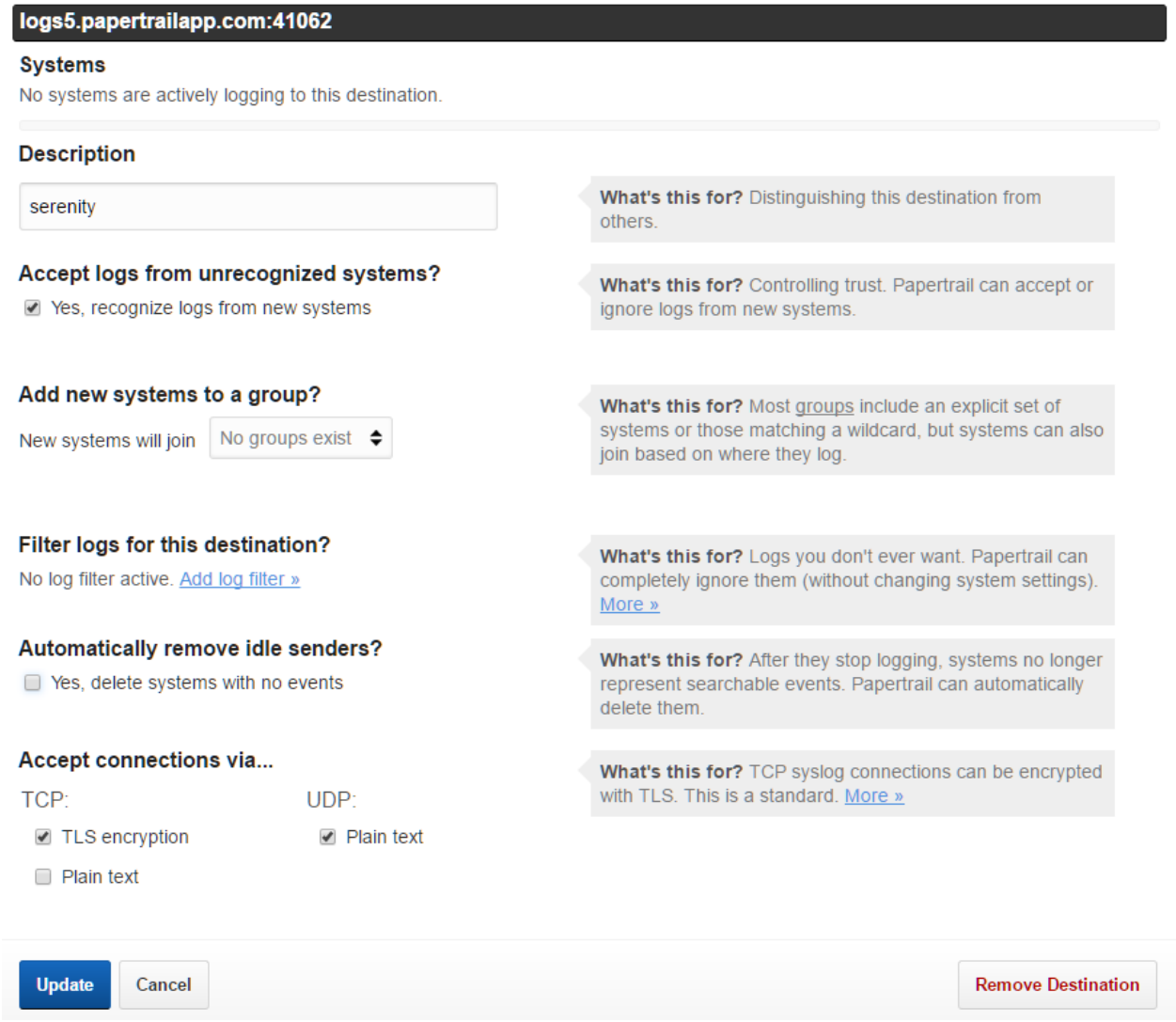
To enable visibility into the log files produced by your applications within docker, we have to introduce a backing service to process these logs, store them and give us visibility into what's going on. The main difference between this backing service, and let's say - a database is the way it's interfaced directly with docker, instead of adding new responsibilities into your application stack.

Logging Docker to Papertrail

I'm going to set up Papertrail[93] to enable inspection of our docker logs. If you sign up with he link here, it contains a referral code which gives us both 50MB volume extra each month for the next 12 months. To send docker logs to Papertrail, we will also set up gliderlabs/logspout[94].

In order to send logs to Papertrail, we have to create a "Log Destination" first. I'm going to name mine "serenity" - the name of my development VM. If you're wondering, yes, the name is a reference to Firefly[95].

[93] https://papertrailapp.com/?thank=7dedeb
[94] https://github.com/gliderlabs/logspout
[95] http://www.imdb.com/title/tt0303461/

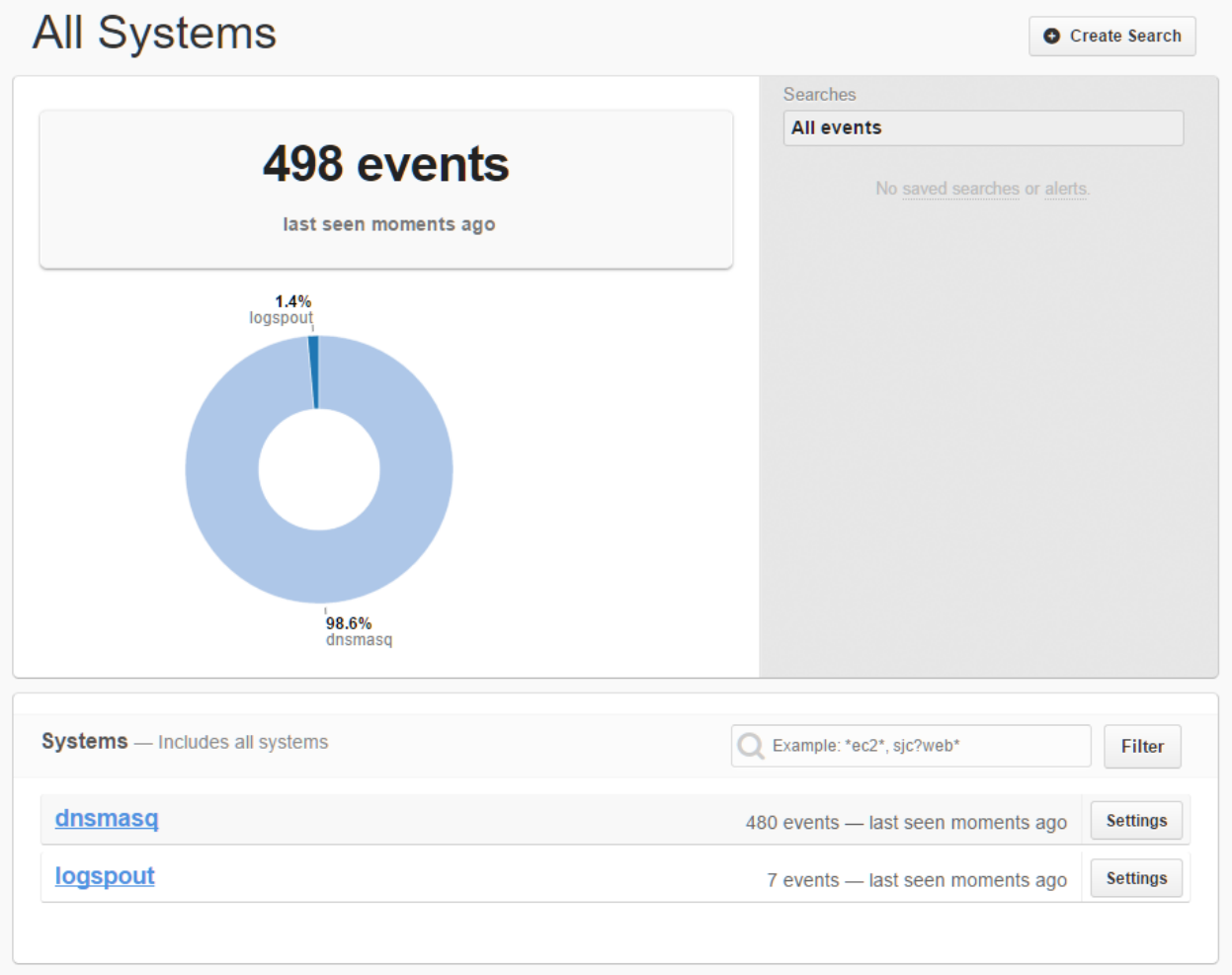

logs5.papertrailapp.com:41062
Systems
No systems are actively logging to this destination.
Description
serenity
What's this for? Distinguishing this destination from others.
Accept logs from unrecognized systems?
Yes, recognize logs from new systems
What's this for? Controlling trust. Papertrail can accept or ignore logs from new systems.
Add new systems to a group?
New systems will join
No groups exist
What's this for? Most groups include an explicit set of systems or those matching a wildcard, but systems can also join based on where they log.
Filter logs for this destination?
No log filter active. Add log filter »
What's this for? Logs you don't ever want. Papertrail can completely ignore them (without changing system settings). More »
Automatically remove idle senders?
Yes, delete systems with no events
What's this for? After they stop logging, systems no longer represent searchable events. Papertrail can automatically delete them.
Accept connections via...
TCP:
TLS encryption
Plain text
UDP:
Plain text
What's this for? TCP syslog connections can be encrypted with TLS. This is a standard. More »
Update
Cancel
Remove Destination

Creating a new log destination

And then we just run a logspout container on the host we want to collect logs from:

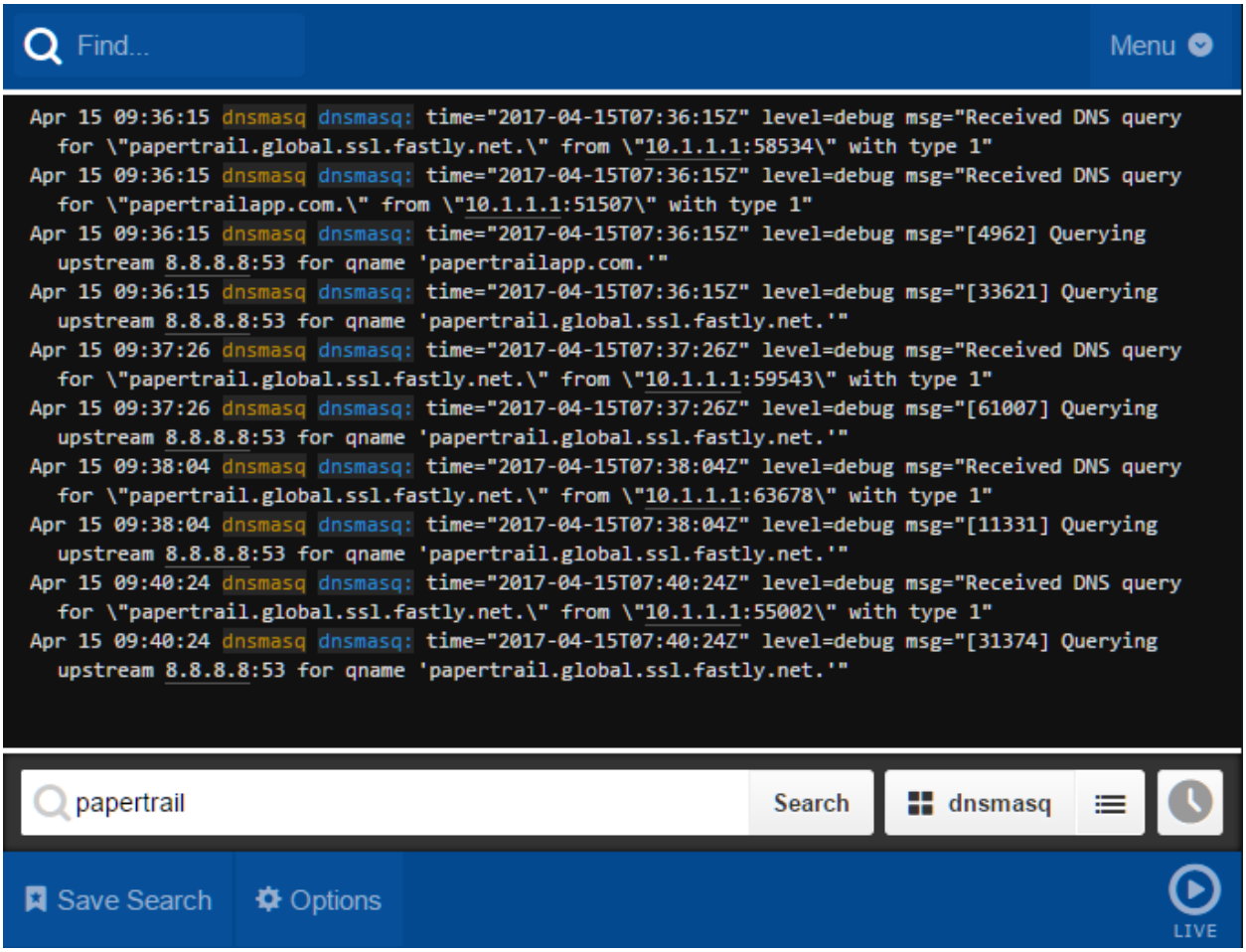
```
# docker run --restart=always -d -h logspout \
  -v=/var/run/docker.sock:/var/run/docker.sock \
  gliderlabs/logspout  \
   syslog://logs5.papertrailapp.com:41062
```

You should replace `logs5.papertrailapp.com:41062` with the log destination that you created. The containers that produce output on stdout will immediately be visible on the Dashboard.



All systems overview

The systems are docker containers, which are registered with their hostnames. As we ran our logspout service with `-h logspout`, it's visible under that name. If we didn't do that, there will be only the hostname reported by the container, usually something like `6feb7e9003e4` (the container ID).


Find...
Menu
Apr 15 09:36:15 dnsmasq dnsmasq: time="2017-04-15T07:36:15Z" level=debug msg="Received DNS query for \"papertrail.global.ssl.fastly.net.\" from \"10.1.1.1:58534\" with type 1"
Apr 15 09:36:15 dnsmasq dnsmasq: time="2017-04-15T07:36:15Z" level=debug msg="Received DNS query for \"papertrailapp.com.\" from \"10.1.1.1:51507\" with type 1"
Apr 15 09:36:15 dnsmasq dnsmasq: time="2017-04-15T07:36:15Z" level=debug msg="[4962] Querying upstream 8.8.8.8:53 for qname 'papertrailapp.com.'"
Apr 15 09:36:15 dnsmasq dnsmasq: time="2017-04-15T07:36:15Z" level=debug msg="[33621] Querying upstream 8.8.8.8:53 for qname 'papertrail.global.ssl.fastly.net.'"
Apr 15 09:37:26 dnsmasq dnsmasq: time="2017-04-15T07:37:26Z" level=debug msg="Received DNS query for \"papertrail.global.ssl.fastly.net.\" from \"10.1.1.1:59543\" with type 1"
Apr 15 09:37:26 dnsmasq dnsmasq: time="2017-04-15T07:37:26Z" level=debug msg="[61007] Querying upstream 8.8.8.8:53 for qname 'papertrail.global.ssl.fastly.net.'"
Apr 15 09:38:04 dnsmasq dnsmasq: time="2017-04-15T07:38:04Z" level=debug msg="Received DNS query for \"papertrail.global.ssl.fastly.net.\" from \"10.1.1.1:63678\" with type 1"
Apr 15 09:38:04 dnsmasq dnsmasq: time="2017-04-15T07:38:04Z" level=debug msg="[11331] Querying upstream 8.8.8.8:53 for qname 'papertrail.global.ssl.fastly.net.'"
Apr 15 09:40:24 dnsmasq dnsmasq: time="2017-04-15T07:40:24Z" level=debug msg="Received DNS query for \"papertrail.global.ssl.fastly.net.\" from \"10.1.1.1:55002\" with type 1"
Apr 15 09:40:24 dnsmasq dnsmasq: time="2017-04-15T07:40:24Z" level=debug msg="[31374] Querying upstream 8.8.8.8:53 for qname 'papertrail.global.ssl.fastly.net.'"
papertrail
Search
dnsmasq
Save Search
Options
LIVE

The container logs cn be filtered

When you click on a system, you will get a live view of the systems events, similar to running `docker logs -f [container]` on the host. Papertrail gives you a search interface with which you can filter the logs. Since I run a janeczku/go-dnsmasq[96] to provide me with a development DNS service, I searched for all the log entries which contained the word `papertrail`.

With Papertrail it's now possible to receive alerts to various services, as soon as an issue is detected from your logs. I can literally set up an alert whenever I use Facebook to send me an SMS that I should stop wasting my time. I'm sure it's not what you had in mind, but it just shows that alerting isn't always about system failure.

[96] https://github.com/janeczku/go-dnsmasq

Logging as a backing service

Papertrail can also be used as a backing-service in line with Chapter IV., which means you could send data to Papertrail directly from your Go application. While I wouldn't recommend it, it does have it's use - like splitting system messages away from the access log but still having them logged to Papertrail, or providing a centralized logging pipeline that could then forward individual events to different services like Loggly[97], StatHat[98], DataDog[99] and others.

As I already mentioned structured logging, a short introduction. Usually logs are written something along these lines:

```
127.0.0.1 - GET /api/request HTTP/1.0 200 "curl" ...
```

This is just a fragment of a HTTP access log, which I wrote from memory, so don't quote me exactly verbatim in this case. The problem with this kind of log is obviously that it's not machine friendly. A structured log entry would be something like this:

```
{ "ip": "127.0.0.1", "method": "GET", "uri": "/api/request", "protocol": "HTTP/1\
.0", ...
```

This JSON payload is something that can be directly read and consumed by applications. The structure of the log entry also makes it easier for applications to extend log entries with additional fields, without worry of breaking a parser or providing the final entry fields in a different order.

For creating structured logs, I recommend using apex/log[100]. It already comes bundled with a papertrail handler, which we will use for our needs.

Multiple logging handlers

```
package main

import (
  "errors"
  "os"
  "time"

  "github.com/apex/log"
  "github.com/apex/log/handlers/multi"
  "github.com/apex/log/handlers/papertrail"
```

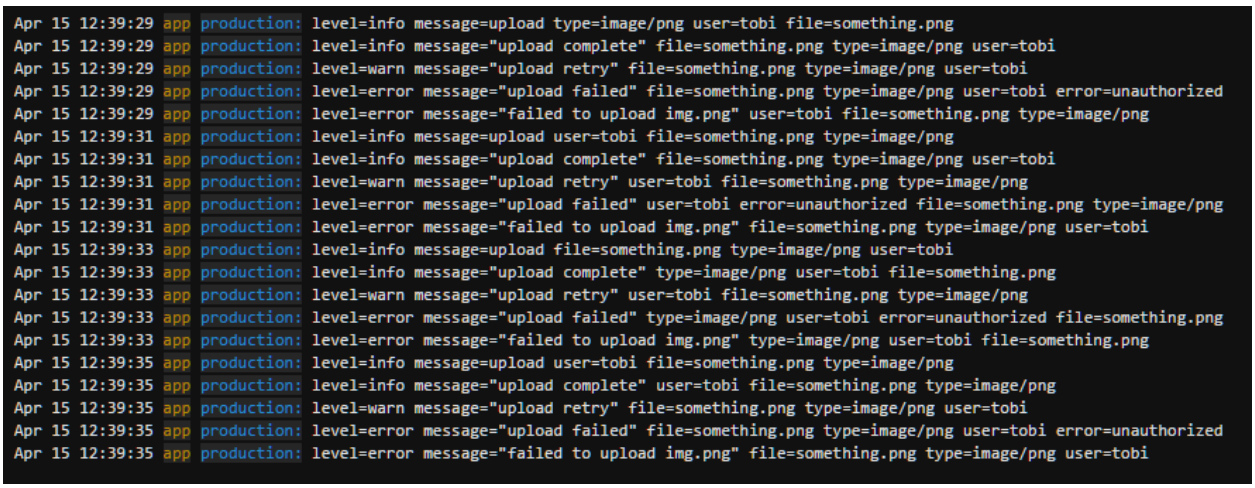
[97] https://loggly.com
[98] https://www.stathat.com
[99] https://www.datadoghq.com
[100] https://github.com/apex/log

```
  "github.com/apex/log/handlers/text"
)

func main() {
  handlerText := text.New(os.Stdout)
  handlerPapertrail := papertrail.New(&papertrail.Config{
    Host:     "logs5",
    Port:     41062,
    Hostname: "app",
    Tag:      "production",
  })
  handler := multi.New(handlerPapertrail, handlerText)

  log.SetHandler(handler)

  ctx := log.WithFields(log.Fields{
    "file": "something.png",
    "type": "image/png",
    "user": "tobi",
  })

  for range time.Tick(time.Second * 2) {
    ctx.Info("upload")
    ctx.Info("upload complete")
    ctx.Warn("upload retry")
    ctx.WithError(errors.New("unauthorized")).Error("upload failed")
    ctx.Errorf("failed to upload %s", "img.png")
  }
}
```

I adjusted the example to provide logging to two destinations - the Stdout as required by this chapter, as well as a backing service with Papertrail. The log package provides several handlers, and I used the `multi` handler to combine logging to `text` as well as `papertrail` handlers. When we run the example we get this printed to the stdout:

```
# ./run
  INFO[0006] upload                        file=something.png type=image/png user=to\
bi
  INFO[0006] upload complete               file=something.png type=image/png user=to\
bi
  WARN[0006] upload retry                  file=something.png type=image/png user=to\
bi
 ERROR[0006] upload failed                error=unauthorized file=something.png typ\
e=image/png user=tobi
 ERROR[0006] failed to upload img.png     file=something.png type=image/png user=to\
bi
... output repeats every 2 seconds
```

And of course the logs get sent to Papertrail as well. There we can filter them by searching for `level=error` or similar searches.



Application logs on Papertrail

By writing your own handlers you can distribute log events from your app to several backing services. Of course, you could use the AWS Kinesis, Elastic search, or any other provided handlers.

XII. Admin processes - Run admin/management tasks as one-off processes

The requirement of 12 Factor apps to run admin/management tasks as one-off processes might seem redundant. Never the less programmers often get caught in the trap of providing these services with tight coupling to the application. The tight coupling prevents running these tasks without an active application and hinders usability of administrative functions just by requiring the complete environment to be set up.

The main benefit of separating administration tasks into one-off processes is that they may operate on certain parts of your infrastructure, without requiring the complete application or service environment to be present. These tasks may be any of the following:

1. Running a database migration,
2. Console access to your containers,
3. Running one-time scripts from your app

Running backups

When it comes to one-off processes, none are more standard than doing database or file backups. When it comes to docker, there's only a slight difference between two possible approaches.

Run one-off tasks inside running containers

Percona is the creator of the MySQL backup tool xtrabackup[101]. The xtrabackup tool requires access to the filesystem where your MySQL instance is running, meaning it has to run from within the container.

Typically, you'd run the following command to perform a full backup:

```
1 xtrabackup --backup --target-dir="/data/backups/mysql/"
```

And when it comes to doing the same within a running container:

```
1 docker exec -it mysql xtrabackup --backup --target-dir="/data/backups/mysql"
```

This is, of-course, provided that you are running a mysql container that contains the xtrabackup tool. Another tool provided is `innobackupex`. This is a script provided by Percona that acts as a wrapper around xtrabackup program. The script enables you to perform point-in-time backups of InnoDB, XtraDB or MyISAM tables and data.

```
1 docker exec -it db innobackupex --user=root --password="[default]" /data/`date +\
2 %Y-%m-%d-%H_%M_%S`
```

This is be a good way to create a complete database snapshot which will be stored in a folder, the name of which is the date/time of when the snapshot was created. All you need to do is make sure that `/data` is a volume mount to some kind of persistent storage.

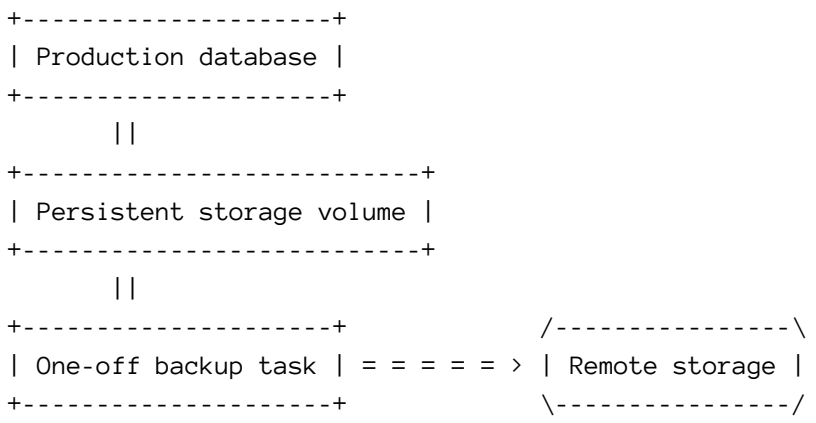
> Technically, a volume mount is not a requirement. You can copy the data to a cloud storage provider. One of my favourites is online.net's C14[a] - they expose ssh/rsync access where you can drop files, and they archive it after about one week.
>
> [a] https://www.online.net/en/c14

[101] https://www.percona.com/doc/percona-xtrabackup/2.4/index.html

Run one-off tasks outside running containers

When it comes to simple filesystem storage - you're backing up something that's essentially simple - as such, you could use `rsync` from outside of the container, copying the volume mount which you have.

As cloud storage may typically be shared between multiple containers, the better way is to provide your persistent storage volume to an additional temporary container. As the backups are being made, this container doesn't steal resources from production instances, and is removed when the program completes.



As my poorly drawn ASCII graph indicates, both the production database and the one-off backup task should have access to the persistent storage. The backup task may run on the same docker container infrastructure, but using additional resources (and not stealing some resources away from the production database, which might become a performance/stability issue in long-running CPU/IO intensive tasks).

Console access to your containers

Containers expose their internals via the `docker exec` command. With it, you are free to start any command in your container. This allows you to run tasks against production instances of your containers, possibly while not targeting a specific instance but an service group. With this, one can typically inspect and extend containers that run in production, for example:

```
1 docker exec -it app-container /bin/bash
```

The above command would execute the application `/bin/bash` inside your container, providing you a remote access shell, much like ssh does when you use it to access a remote server. Depending on what you run in the container, or what you're trying to inspect, you may use any of the common system tools, provided that you've added them to the container.

And in case you forgot to add them - no worries. It's possible to add software to your running container instance - just keep in mind, that this falls under "making things work", and not under "best practice". Containers should be considered ephemeral, or for lack of a better term, "cattle" - they can be killed off and replaced, or at leas that's the goal. When you modify them as they are running, it can be useful during their lifetime, but as soon as they die you'd have to start again.

```
1 app-container$ apt-get update ; apt-get install -y telnet tcpdump
```

The above command would work just as it would in a running server - it would update the package list from the APT repositories listed under /etc/apt, and would install telnet and tcpdump, which you can use until the container is reaped.

> This is perhaps far from best practice. I'm pretty sure that people with 10 thousands of running containers never do such a thing in production. But for development purposes (or for a very critical issue which needs diagnostic tools), it's good to know how to approach the diagnosis of the problem.

Why does this work?

This is possible because of the magic of the 'overlay' filesystem, which docker employs when running their containers. To paraphrase how it works:

1. When you start a container, the contents of the docker image are copied,
2. Volume mounts if any are created in-lieu of certain locations,

3. When you add, modify or delete files outside of the volumes, an additional filesystem layer is created,
4. Docker uses 'overlay' to "merge" the original image layer and the new filesystem layer into a consistent view (the running container filesystem)

Imagine if you will, running a database instance without a volume mount - it is possible, but due to the restrictions of the overlay filesystem, the performance will be bad. In case you accidentally `docker rm` your container - you would lose all your data. The list of "caveat emptor" continues.

However, if you need to inspect a running container with `tcpdump`, installing it is not bad-practice per se, but sheer ingenuity on the part of the sysadmin/ops guy. After all - if you're experiencing issues, you don't want to be locked into the running container; rebuilding it might take at least a few minutes and any issues you might be experiencing might go away after restarting the container.

A note on keeping container size down

When building docker images there are generally three approaches when it comes to packaging Go binaries. You can use `scratch` which will give a uninspectable container with zero software but tiny size. You can use `alpine` which will only add about 5MB but give you the option to install additional software like curl, netdata, telnet and others - enabling some sort of inspection of the running container, while keeping size down. And the third option: `debian` or `ubuntu` flavour images. These usually weigh in at about 100MB, which may mean a performance penalty when it comes to running containers on low-end hardware, for example.

Keep in mind that choosing a smaller base image to work with has performance implications as well. When docker starts a container, it makes a copy of the image into a new, unique location on disk. If the image starts with `debian`, it would mean that it has to copy 100MB to start with. It might mean next to nothing on modern server hardware, but consider your image might have `java` and weighs in at 1GB. Consider that you may run docker on less than state of the art hardware, and it's possible to run it on a Raspberry PI which gives you 10-20MB/s read/write speed on the SD card, and a solid few seconds to start a 100MB container.

Another thing to keep in mind might be that sometimes, other people (or new servers) might need to download an updated image from Docker Hub or from your internal registry server. Let's say your fleet is relatively small with 100 nodes, and gigabit connectivity between them - a 100MB image will download for about a second, needing about 2 minutes to update the cluster completely. If your container would be 1GB it will take about 20 minutes. If your network speed would go down from gigabit to 100mbit, you're looking at several hours of updates. This becomes a basic bandwidth/size/speed calculation that has a real-world impact.

My recommendations:

Use Alpine: Even when you're packaging Go applications, use the `alpine` base image, and not `stratch`. Alpine bundles their package manager, `apk`, which you can use to install things like `bash`, `telnet`, `tcpdump`, `jq` or other command line tools that may assist you in debugging almost anything

you can think of. And `scratch` is missing some usable defaults, like the public HTTPS keyring so you can fetch resources from public https services (like various APIs).

Don't have same build/run environments: Don't run a full build environment in production. When it comes to `golang` images, they are the complete compiler and allow you not just to run any Go program, but to build it as well. The built Go binary would usually weigh about 10-20MB, while the compete build system would weigh at least 200MB. Using it would also go against the 12 Factor App guidelines - build and run stages should be *separated*.

> When it comes to scripting languages like *PHP* or *Node*, the distinction between build and run stages becomes blurred, as it is a scripting and not a programming language with a compiler. In such cases there are development-only tools like `composer`, or `npm` that provide software based libraries which your project includes in the run stage, but installs in the build stage. Node.js actually has a "compiler" called Nexe[a] that produces an executable binary out of your Node application. It's a good way to shave several 100MB off your application image.
>
> [a] https://github.com/nexe/nexe

Orchestrating your management tasks

Depending on what application you manage within a container, the one-off processes might need the running environment of the container, or they may spin up and tear down containers as necessary. For example: running unit tests against your code does not require a running application container - you can use your application image to start a new isolated container, running tests completely independently. This way, the one-off process runs in an identical environment as your application.

Testing your software falls under the "build" stage of chapter V., however you might want to run unit tests manually during development, and not just when you push code. The development environment should contain all the elements of the production environment (Chapter X., prod/dev parity).

A trivial example of a unit test that uses docker to orchestrate additional containers would be similar to this:

```
#!/bin/bash
echo "Running tests in docker container"
docker rm -f app-redis app-memcached app-nutcracker
docker run --net=party -d --name=app-redis redis
docker run --net=party -d --name=app-memcached memcached
docker run --net=party -d --name=app-nutcracker -v $(dirname $(readlink -f $0))/\
nutcracker:/etc/nutcracker registry/nutcracker
docker run --net=party --rm -it -v $CWD:/app -w /app --dns 8.8.8.8 registry/app \
.build/build.sh
EXITCODE="$?"
exit $EXITCODE
```

The concrete example is of a `run.sh` script that launches unit tests in the `registry/app` container. The container needs a couple of things to test - most notably running instances of `redis`, `memcached` and `nutcracker` (or twemcached if you prefer). The exit code of the unit tests is caught and passed on. In case of non-zero exit codes, your CI solution will stop the build process - and in the case of development, notify the developer of the failing tests directly.

> This kind of test can not be used within the Dockerfile. We can reasonably say, that this kind of test that requires orchestrating docker resources falls into an extended "build" stage of the application, which includes but isn't exclusive to what you have in the Dockerfile.